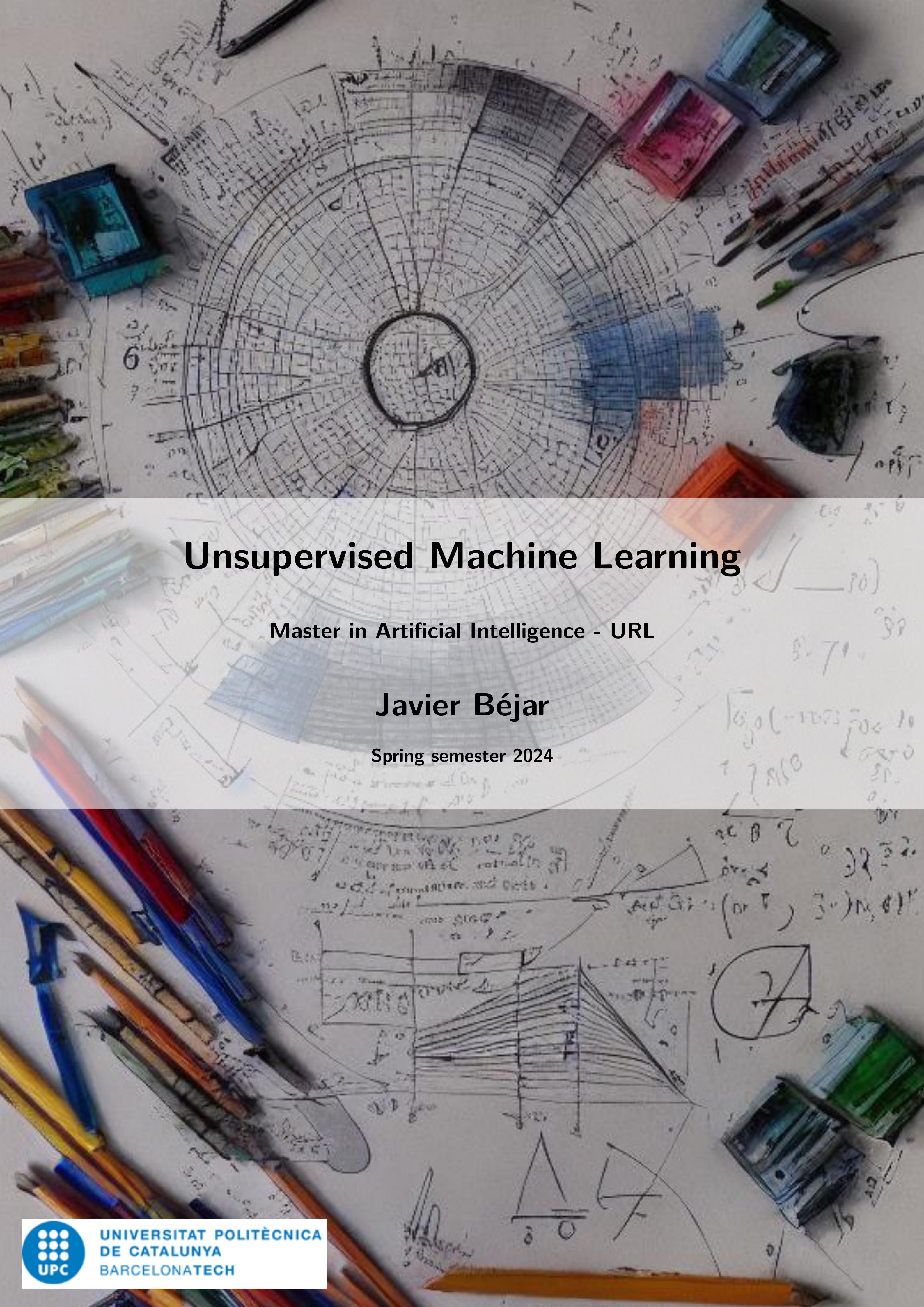


Unsupervised Machine Learning

Master in Artificial Intelligence - URL

Javier Béjar

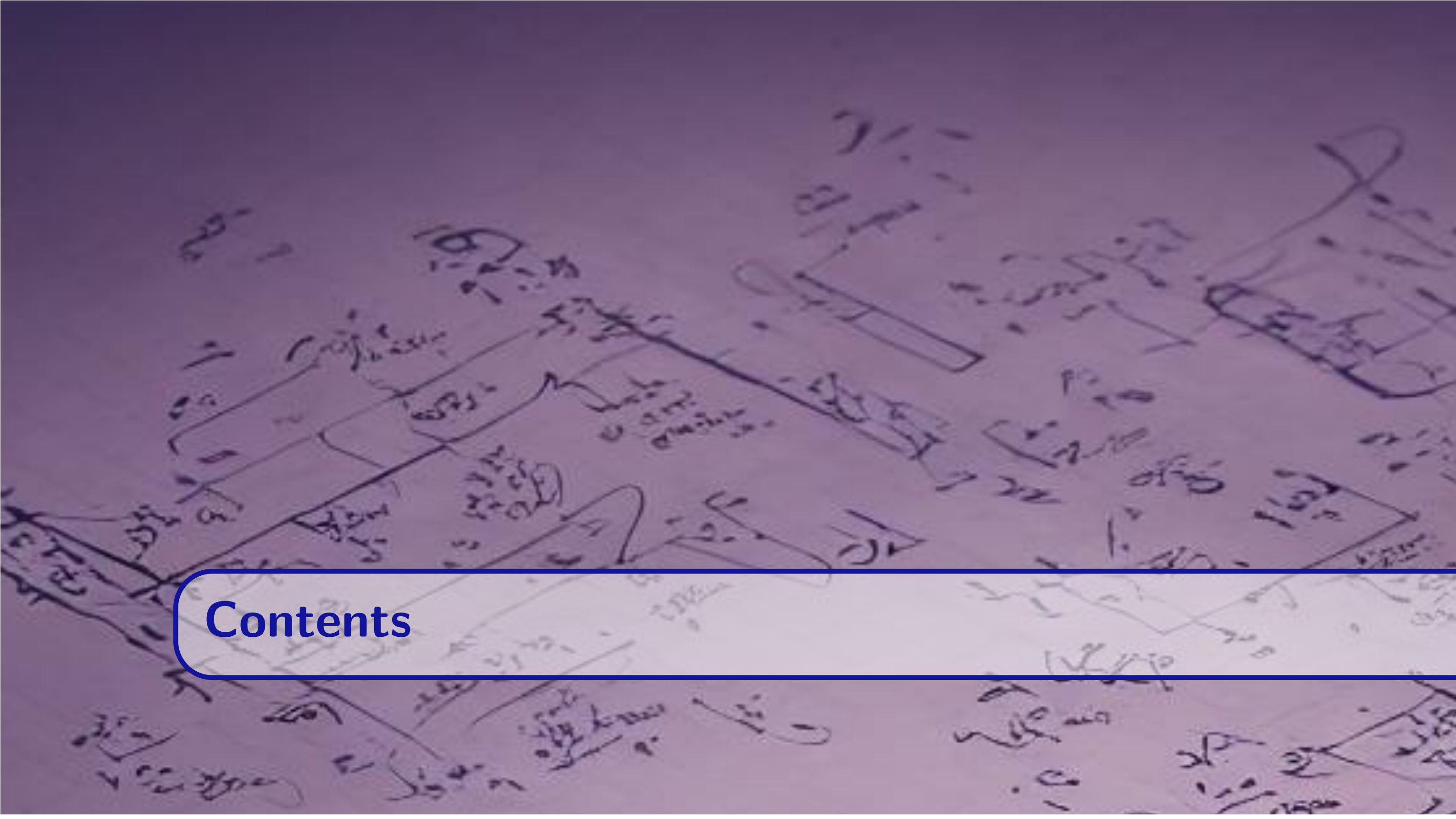
Spring semester 2024

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Cover and chapter headings generated with Stable Diffusion

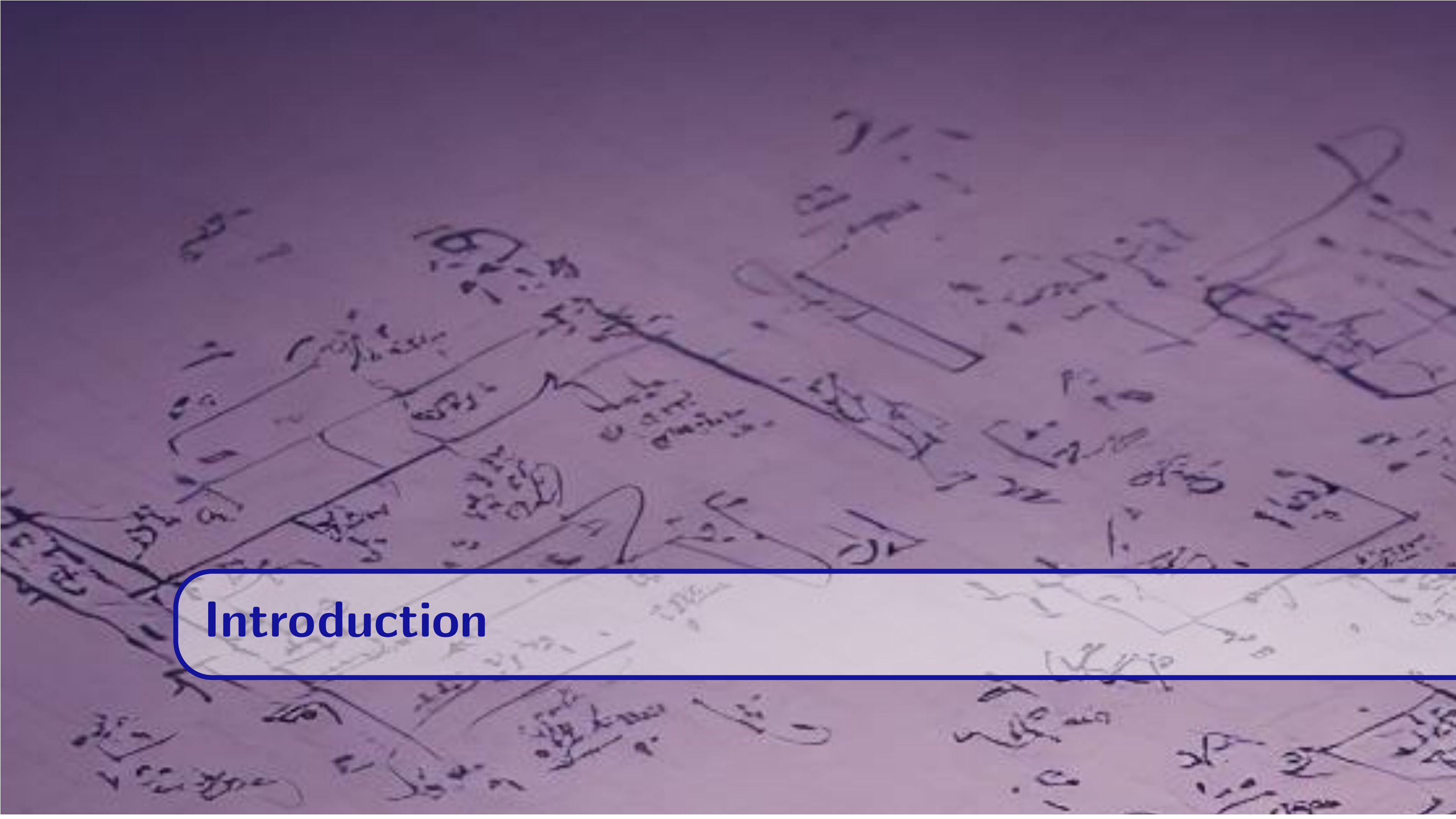
First edition, February 2018

This edition, February 2024



Contents

II Clustering



Introduction

These are the lecture notes for the course Unsupervised Learning (UL). The idea of these notes is to facilitate following the topics of the course and to allow a more dynamic work in class. It is necessary that you read the material corresponding to each scheduled class beforehand, so you can use and practice the different techniques to be explained during the class.

Most of the chapters of this lecture notes have accompanying Jupyter Notebooks examples that you can find in the web page of the course with also the datasets that are used. The examples use different python libraries implementing the algorithms explained during the lectures, mainly from the `scikit-learn` library, that is available for install in any python distribution using pip. Some methods are implemented in the library kemlglearn that you can find in `https://github.com/bejar/kemlglearn`. This library can also be installed using pip (`pip install kemlglearn`).

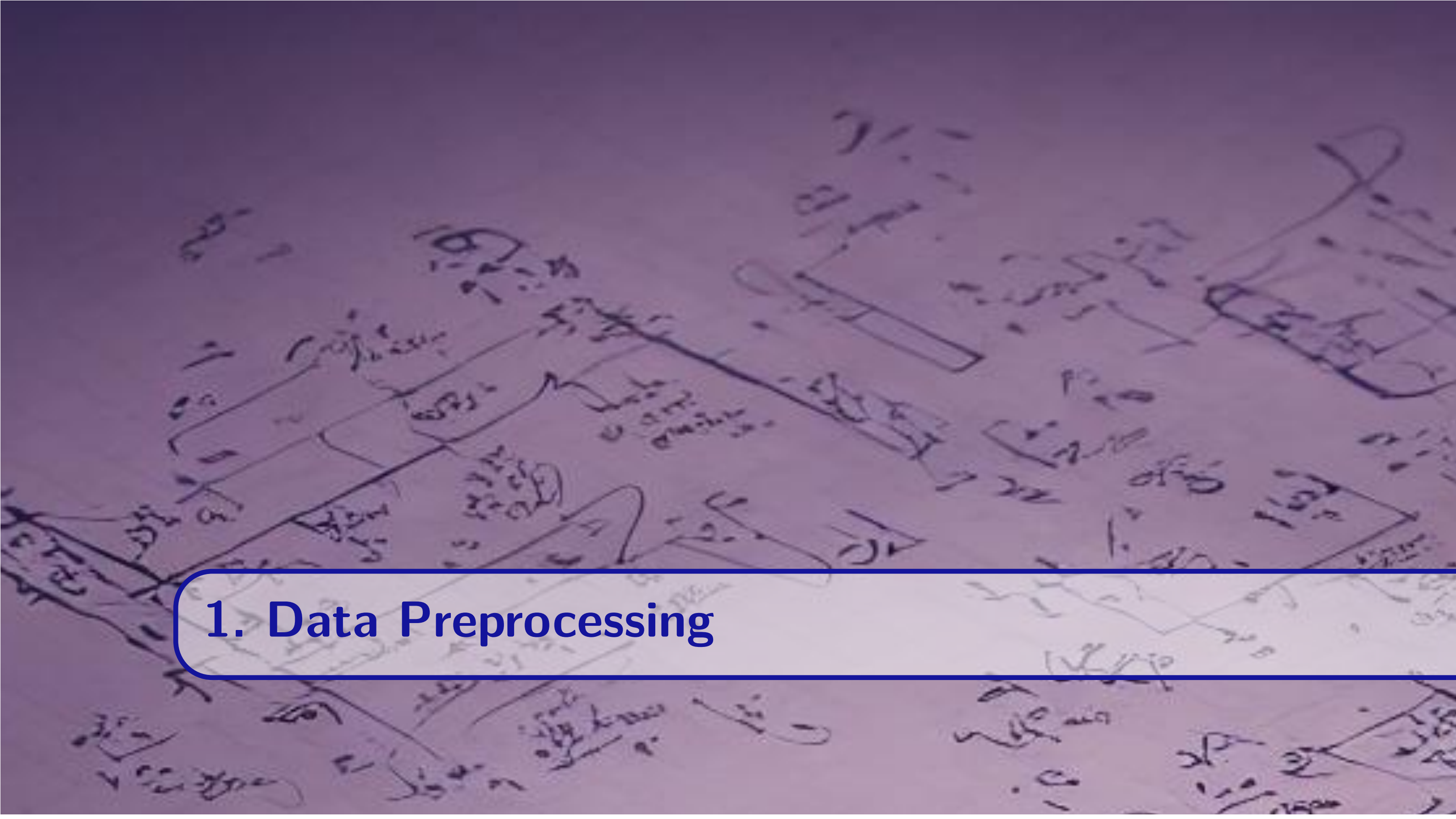
To use the notebooks you will have to install on your computer the libraries that appear on the first cell of each notebook (you just need to uncomment the code). Other possibility is to use Google's Collaboratory. You will find some dependencies already installed, and you will only need to upload the notebooks, but you will have some memory and cpu limitations.

Also, you can install Docker (`https://www.docker.com/`) in your machine and the data-science notebook (`https://hub.docker.com/r/jupyter/datascience-notebook/`) image that contains jupyter notebook, and most of the software that you need.

Every chapter has also a section with the results of its corresponding notebooks with some additional comments.

I

Preprocessing



1. Data Preprocessing

1.1 Datasets

Each domain has its own characteristics that have to be represented accordingly to their semantics. This representation will result on a set of attributes that describe what interest us from the domain, and what we want to use for the discovery process. According to this representation we can categorize the different kinds of datasets in two groups.

The first kind are those that can be represented as a *flat structure* of features, we will call them *unstructured* datasets. These can be described using a list of attributes and an attribute-value matrix. We assume that there is no structural/semantic relationship among the different attributes that describe the data (or we are not interested on it). There will be only one database table/file of observations, where each example is an independent instance of the problem. The examples will be represented by a set of attributes (discrete, continuous...), where each row corresponds to a realization of the attributes. For example, the table in figure 1.1 shows a dataset described by three attributes: A (binary), B (real) and C (discrete).

For the second kind, the *structured* datasets, the attributes of the examples or the examples themselves are related. In this case, we are more interested in the patterns that the relationships contain. These datasets correspond to sequences and tree/graph structures. In a sequence we

A	B	C	···
1	3.1	green	···
1	5.7	blue	···
0	-2.2	blue	···
1	-9.0	red	···
0	0.3	yellow	···
1	2.1	green	···
⋮	⋮	⋮	⋱

Figure 1.1: Example of the representation of an unstructured dataset

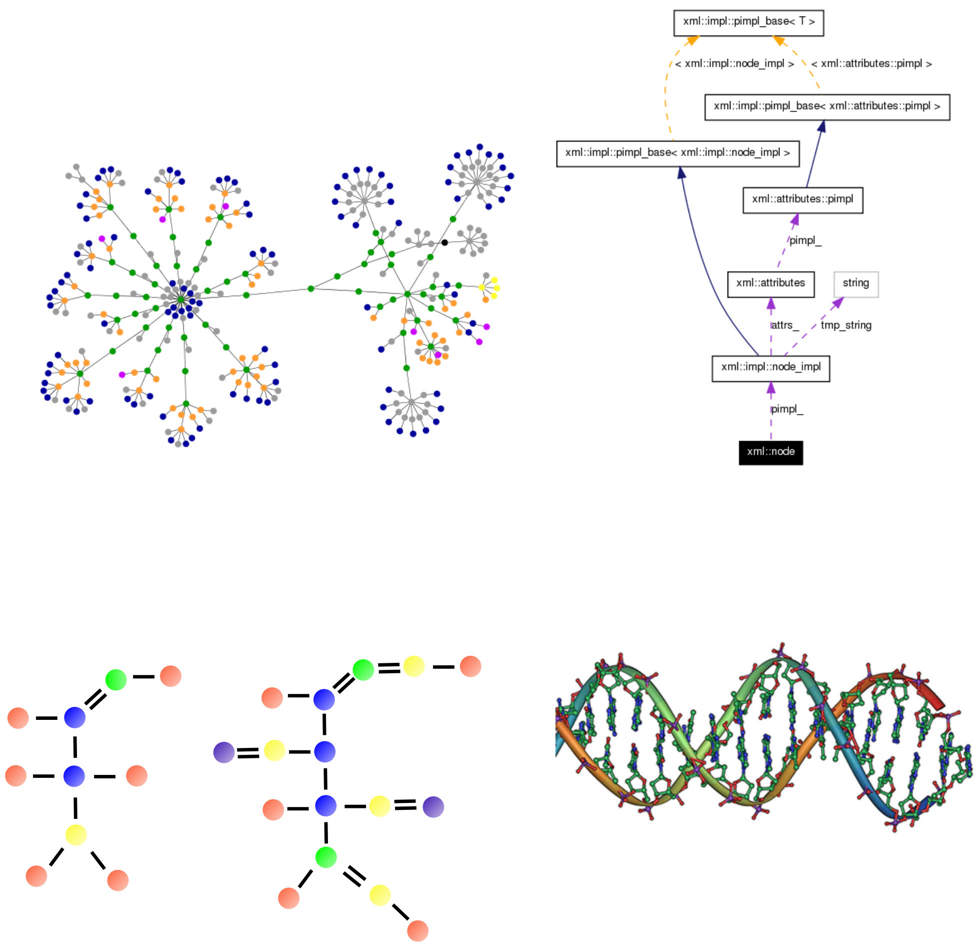

xml::impl::pimpl_base< T >
< xml::impl::node_impl >
< xml::attributes::pimpl >
xml::impl::pimpl_base< xml::attributes::pimpl >
xml::impl::pimpl_base< xml::impl::node_impl >
xml::attributes::pimpl
pimpl_
xml::attributes
string
attrs_
tmp_string
xml::impl::node_impl
pimpl_
xml::node

Figure 1.2: Examples of structured datasets (Social Network, XML graph, Molecules database, DNA sequence)

have an order (total or partial) relationship among the examples of the dataset. This order usually corresponds with time (time series) or to a domain defined precedence like, for instance, the nucleotids order in DNA. This sequence can contain itself unstructured examples, being it a sequence of unstructured data, for instance, text obtained from news feeds or purchases from an online store. It can also be a sequence of complex data, like small sequences interleaved in time, such as click behavior of users in a website, where each user generates a sequence of events, with all that sequences conforming the sequential dataset.

With tree and graph structures we have more complex relationships among the attributes. These kinds of data can also be just an instance, (e.g. a social network) or a set of structured examples (eg. XML documents, graph representation of molecules, a relational database). Figure 1.2 illustrates some examples of these kinds of datasets.

1.2 Data Preprocessing

Usually raw data is not directly adequate for analysis for different reasons. For instance, the quality of the data could not be adequate because of noise in their attribute values, the existence of examples with missing values, or extreme values on their attributes (outliers). Also, sometimes the dimensionality of the data is too high to be efficiently processed because there are too many attributes or too many examples.

The first step before performing any data analysis task is to assess the quality of the data

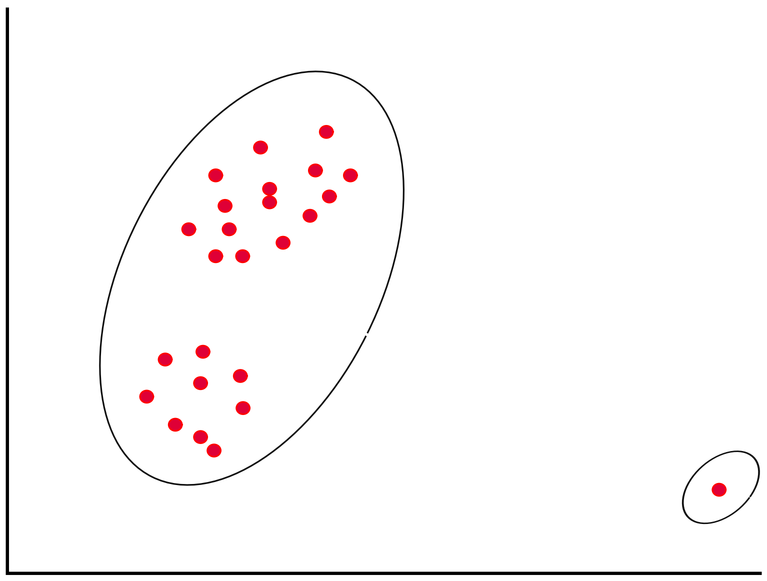


Figure 1.3: Effect of outliers in data partitioning

checking for any of the mentioned problems. Each one of them can be addressed using different techniques and algorithms.

1.2.1 Outliers

Data outliers are defined as examples that have values in some or all of their attributes that deviate significantly from the distribution of the rest of the values of the dataset. These examples can be considered as having extreme values for some of their attributes. The main problem is that their presence in our dataset can have an important impact on the results of some algorithms like, for instance, on data partitioning. This problem is illustrated in figure 1.3), where looking for a partition in two groups of the data, one outlier has been considered one of these partitions.

A simple way of dealing with these data is to remove the examples from the dataset. Alternatively, if the exceptional values appear only in a few of the attributes, the actual values could be ignored and treated as missing values.

An excellent short review on this topic can be found in [68], and for a more in deep introduction you can consult the book [1]. There are different methods for detecting outliers in data that can be divided into *parametric*, and *non-parametric* methods. The following is a brief summary of some of them.

Parametric methods

Parametric methods assume a specific probability distribution function for the attributes, data can be used to estimate the parameters of that distribution. From the model, a statistical test can be computed to determine the probability of obtaining each specific value in the examples. Those that have a lower than a specific significance value are marked as extreme values and considered outliers.

In the case of a single variable and assuming gaussian data, a Z test or *student's* t test can be used to determine if a value is extreme enough. For the multivariate case, other methods can be applied. The *deviation based* method estimates the reduction in variance obtained by removing a subset of examples. The *angle-based* detection method computes the variance of the angles formed by an example and other pairs of examples. In this case outliers will present a small variance in the distribution of their angles. *Distance based* methods use distance measures like the *Mahalanobis distance* to estimate the distance from the mean of the data for each dimension.

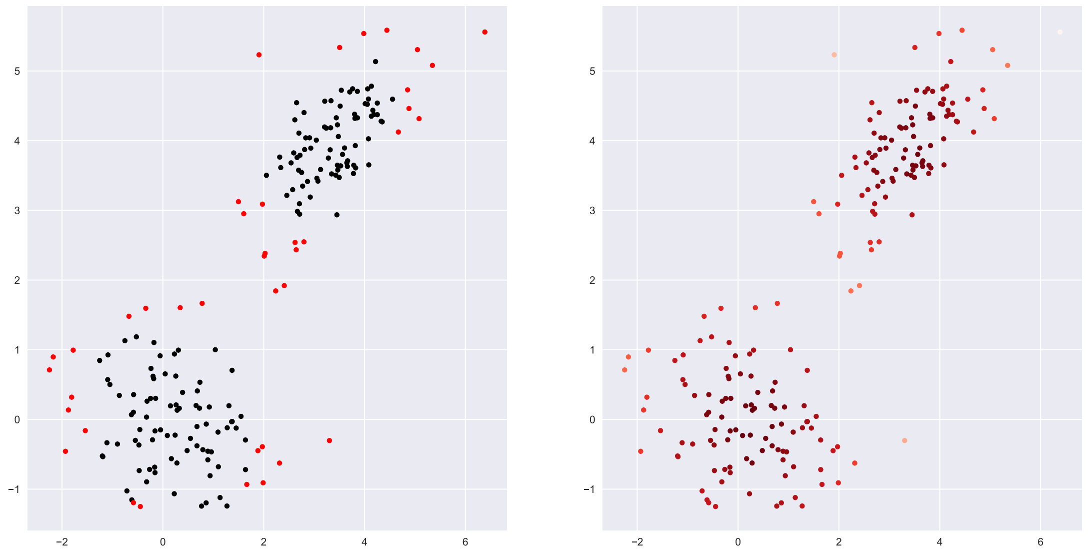


Figure 1.4: Outliers for a dataset with two clusters using the LOF criteria. Left, a 10% of the examples with higher LOF are marked red. Right, the color of the example corresponds with its LOF value

The total distance is distributed as a χ^2 allowing to perform a significance test.

All these methods present the problem that a statistical distribution has to be assumed, they are computationally expensive when the dataset is large and usually only are able to detect outliers that are on the borders of the density of examples.

Non parametric methods

There are many methods that can be used when it is not possible to assume a model for the distribution of the values. They are based on approximations of the probability distribution of the data.

Histogram based methods compute histograms for each attribute individually. The main issue for these methods is to decide an adequate discretization. The grid structure defined can be used to model the sparse regions in the data, discarding the examples that belong to the bins with lower density.

Other possible solution is to use kernel functions and *kernel density estimation* (KDE) to compute a local approximation of the probability distribution. With large enough datasets, the estimation converges to the true density function. For this method the issue is to decide an adequate bandwidth for the kernel.

Distance based outlier detection methods use the assumption that the distances of outliers to their k-nearest neighbors are larger than for normal points. A rank or score can be computed using the distances using for instance the distance to the k-nearest neighbors, or the mean distance to all the k-nearest neighbors. A threshold can be used to decide what examples are labeled as outliers. Different strategies can be used to reduce the cost of computing all the pairs of distances, like indexing structures, bounding boxes or heuristics for discarding candidates as early as possible. The main issue with these methods is their sensitivity to data that presents large variations in their density specially local densities.

Density based methods approach the local density problem. The algorithm *Local Outlier*

Factor (LOF) is an example. This method quantifies the *outlierness* of an example adjusting for the variation in the data density.

The LOF method uses the distance of the k-th neighbor ($D_k(x)$) of an example and the set of examples that are inside this distance ($L_k(x)$). The *reachability distance* between two data points ($R_k(x,y)$) is defined as the maximum between the distance $dist(x,y)$ and the y's k-th neighbor distance ($D_k(y)$).

$$R_k(x,y) = \max\{dist(x,y), D_k(y)\} \quad (1.1)$$

This is not a symmetric distance.

The *average reachability distance* ($AR_k(x)$) with respect x's neighborhood ($L_k(x)$) is defined as the average of the reachabilities of the example to its neighbors. The *Local Outlier Factor* of an example is computed as the mean ratio between $AR_k(x)$, and the average reachability of its k neighbors:

$$LOF_k(x) = \frac{1}{k} \sum_{y \in S_k(x)} \frac{AR_k(x)}{AR_k(y)} \quad (1.2)$$

Outliers will have a large LOF value and examples in a dense area will have a LOF value close to 1.

Figure 1.4 represents the selection of outliers from a two clusters dataset. The 10% of examples with larger LOF value are selected (left), the value of the examples' LOF is represented on the right.

1.2.2 Missing Values

Missing values usually appear because of errors or omissions during the data collection. This circumstance has an impact on the quality of the data. The process of substituting these data is called missing values *imputation*.

There are several methods for computing the value to be used to substitute the missing data. These are some of them:

- To use a global constant for all the values (domain/variable dependent)
- To use the mean or mode of the attribute (global central tendency)
- To use the mean or mode of the attribute but only of the k-nearest examples (local central tendency)
- To learn a model for the data (regression, bayesian), and use it to predict the missing values

Each one of them has different computational cost, the first one is $O(1)$ for each missing value. The second one requires first to compute the statistical moments of the variable $O(|examples|)$, and then the cost is constant for each missing value. The third one needs to compute for each example with missing values the k-nearest examples, this involves $O(|examples| \times \log(k))$, and then to compute the moments of the k examples. Finally, the last one involves to learn a model for each variable, the cost will depend on the cost of building the model.

Something that we have to be aware of, is that the statistical distribution of the data is changed by the imputation of missing values. Using the same value for all the imputations will reduce the variance of the attribute, this is what the first two methods do. Only the second method will not affect the mean of the data. The rest will have less impact in the variance. Figure 1.5 represents the result of missing values imputation using the mean and the 1-nn.

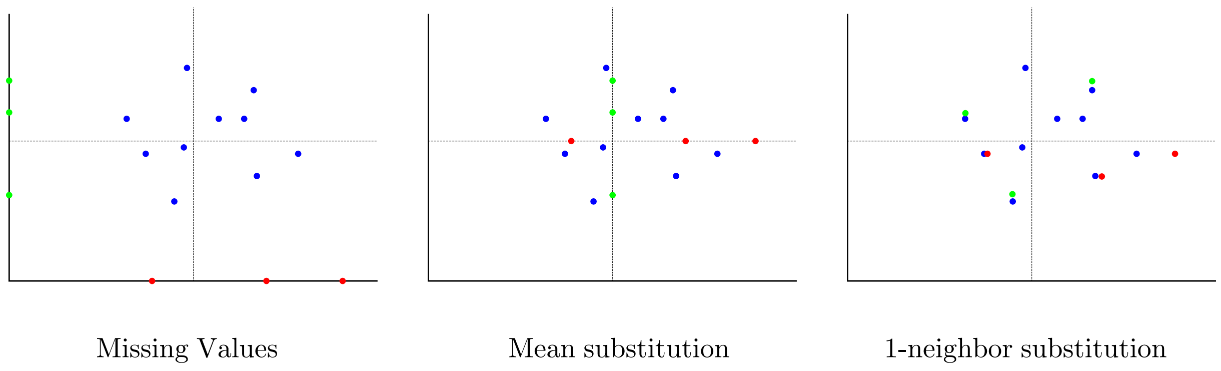


Figure 1.5: Effect of different strategies of missing values imputation

1.2.3 Notebook: Outliers and Missing Values

Now we are going to practice the concepts of the previous subsections. You have this code available as a notebook in the web page of the course. We will use the algorithms that have been described for outlier detection and missing values imputation.

The main libraries that will be used are scikit learn and pyOD that can be installed using pip.

[5]:
```
from sklearn import datasets
from sklearn.neighbors import LocalOutlierFactor
import seaborn as sns
import matplotlib.pyplot as plt
from pyod.models.knn import KNN
from pyod.models.abod import ABOD
from kemlglearn.preprocessing import Discretizer
import warnings
import numpy as np
import matplotlib
warnings.filterwarnings('ignore')
matplotlib.rcParams.update({'font.size':20})
```

Outliers Detection

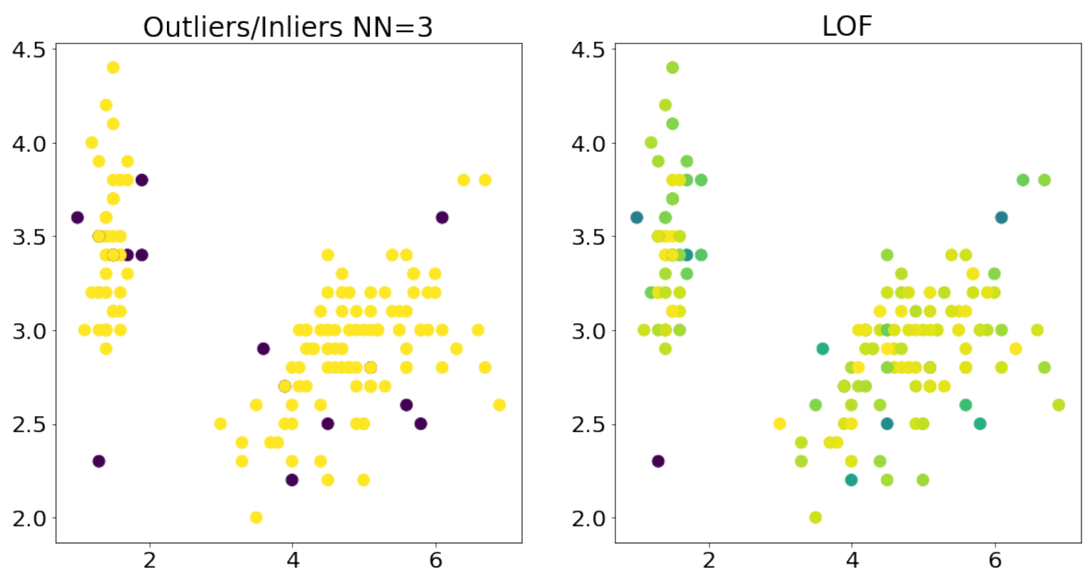
We will use first the **LOF method** to identify a percentage of examples as outliers in the Iris dataset. This method computes an outlierness factor and marks the percentage of examples with higher value.

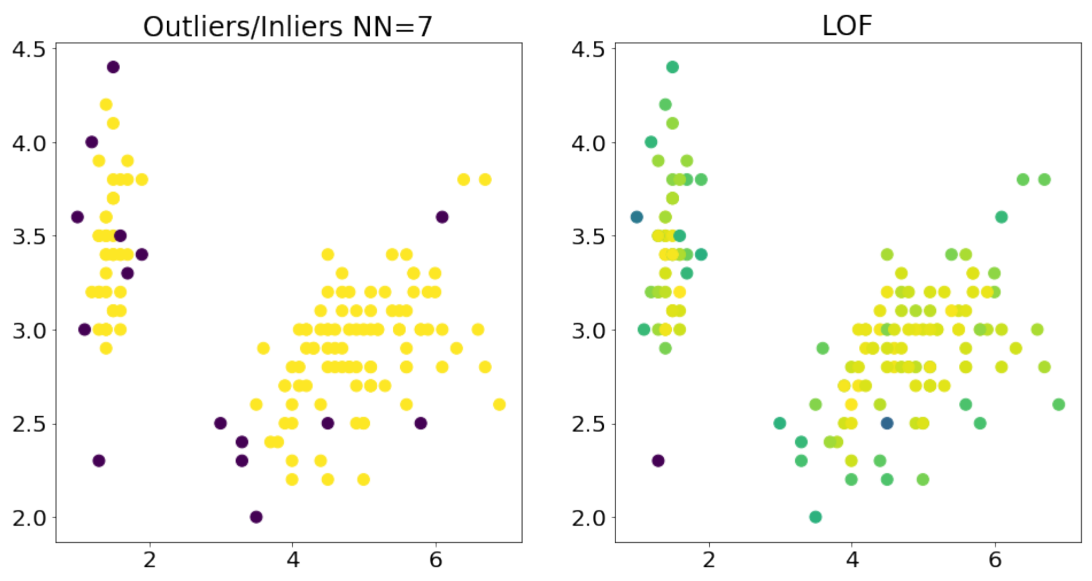
Using different number of neighbors obtains different results because of the change in relationship among the examples and the computation of their densities.

The plots only use two of the variables, so some of the outliers will not seem too far from the more dense parts of the data. This means that probably they are far in the dimensions that are not plotted.

[6]:
```
NEIGHBORS=(3,7)
OUTLIERS=0.1

iris = datasets.load_iris()
iris
for n in NEIGHBORS:
    lof = LocalOutlierFactor(n_neighbors=n, contamination=OUTLIERS)
    labels = lof.fit_predict(iris['data'])
```

```
fig = plt.figure(figsize=(16,8))
plt.subplot(1,2,1)
plt.scatter(iris['data'][:, 2], iris['data'][:, 1],
            c=labels,s=100)
plt.title(f'Outliers/Inliers NN={n}')
plt.subplot(1,2,2)
plt.scatter(iris['data'][:, 2], iris['data'][:, 1],
            c=lof.negative_outlier_factor_,s=100)
plt.title('LOF');
```

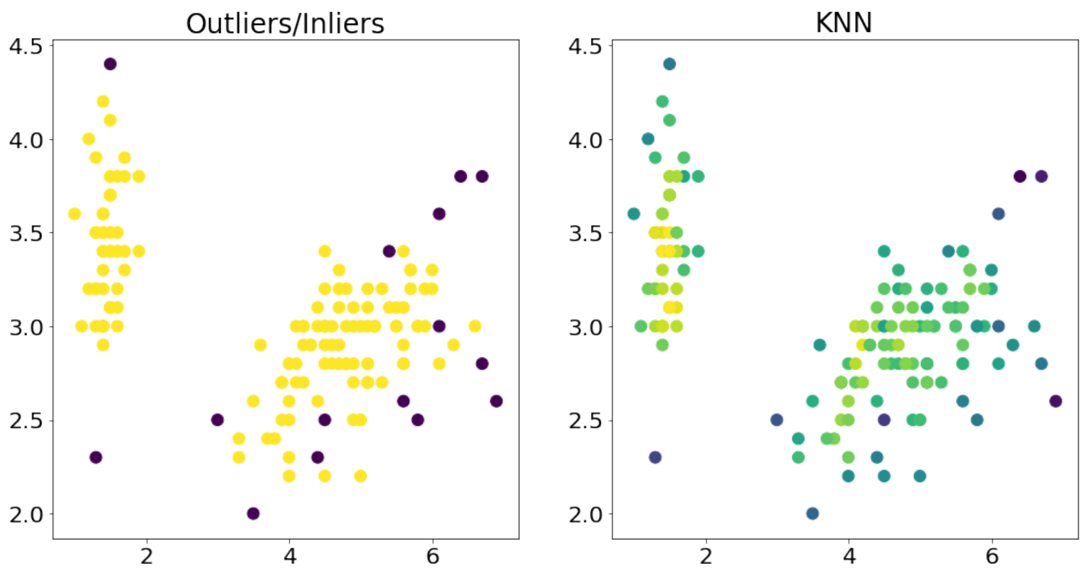




As we can see there are some common outliers using different parameters for the algorithm but depending on the number of neighbors used there are some differences. This has to do with the locality and the density of the locality around the examples that can change if we extend the volume used to compute the outlierness.

This next plot shows the results of an outlier detector based on **K-nearest neighbors**

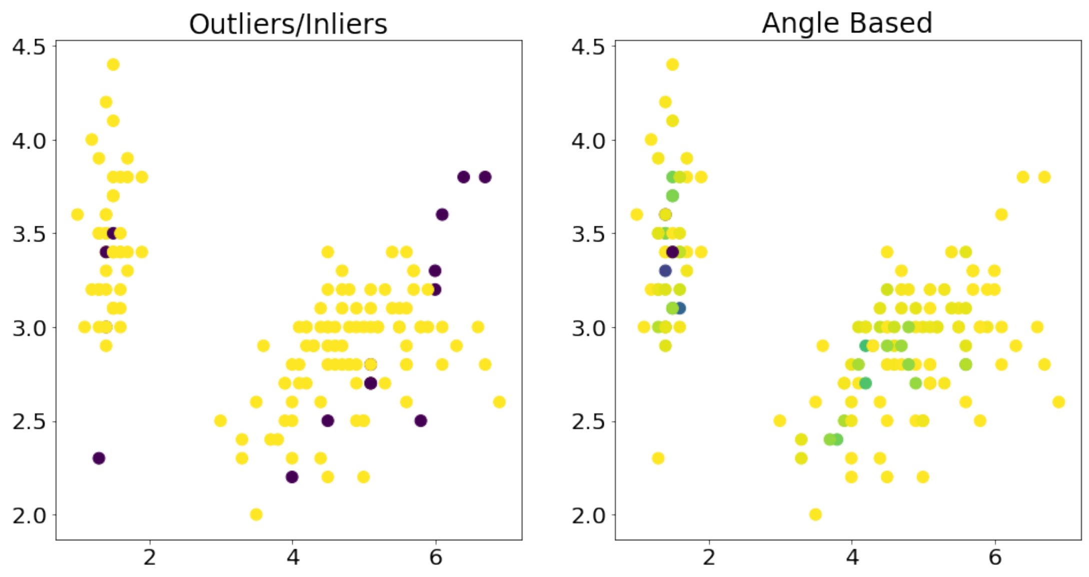
[7]:
```
NEIGHBORS=(3,7)
OUTLIERS=0.1
iris = datasets.load_iris()
knnout = KNN(n_neighbors=NEIGHBORS[0], contamination=OUTLIERS)
labels = 1 - knnout.fit_predict(iris['data'])
fig = plt.figure(figsize=(16,8))
plt.subplot(1,2,1)
plt.scatter(iris['data'][:, 2], iris['data'][:, 1],
            c=labels,s=100)
plt.title('Outliers/Inliers')
plt.subplot(1,2,2)
plt.title('KNN')
plt.scatter(iris['data'][:, 2], iris['data'][:, 1],
            c=1-knnout.decision_scores_,s=100);
```



This algorithm seems to focus more on examples that are on the outskirts of the densities and it is more sensitive to areas of examples with more spread.

Finally, this is an **angle based** outlier detection method

[8]:
```
NEIGHBORS=(3,7)
OUTLIERS=0.1
iris = datasets.load_iris()
about = ABOD(n_neighbors=NEIGHBORS[0], contamination=OUTLIERS)
labels = 1 - about.fit_predict(iris['data'])
fig = plt.figure(figsize=(16,8))
plt.subplot(1,2,1)
plt.scatter(iris['data'][:, 2], iris['data'][:, 1],
            c=labels,s=100)
plt.title('Outliers/Inliers')
plt.subplot(1,2,2)
plt.scatter(iris['data'][:, 2], iris['data'][:, 1],
            c=about.decision_scores_,s=100)
plt.title('Angle Based');
```



Given that the criteria used by each method is different we can not expect to obtain the same result. Usually the outliers are examples isolated or in the border of the densities, so with an specific percentage we can expect to eliminate most of them, but also some examples not too far from the densities will be discarded.

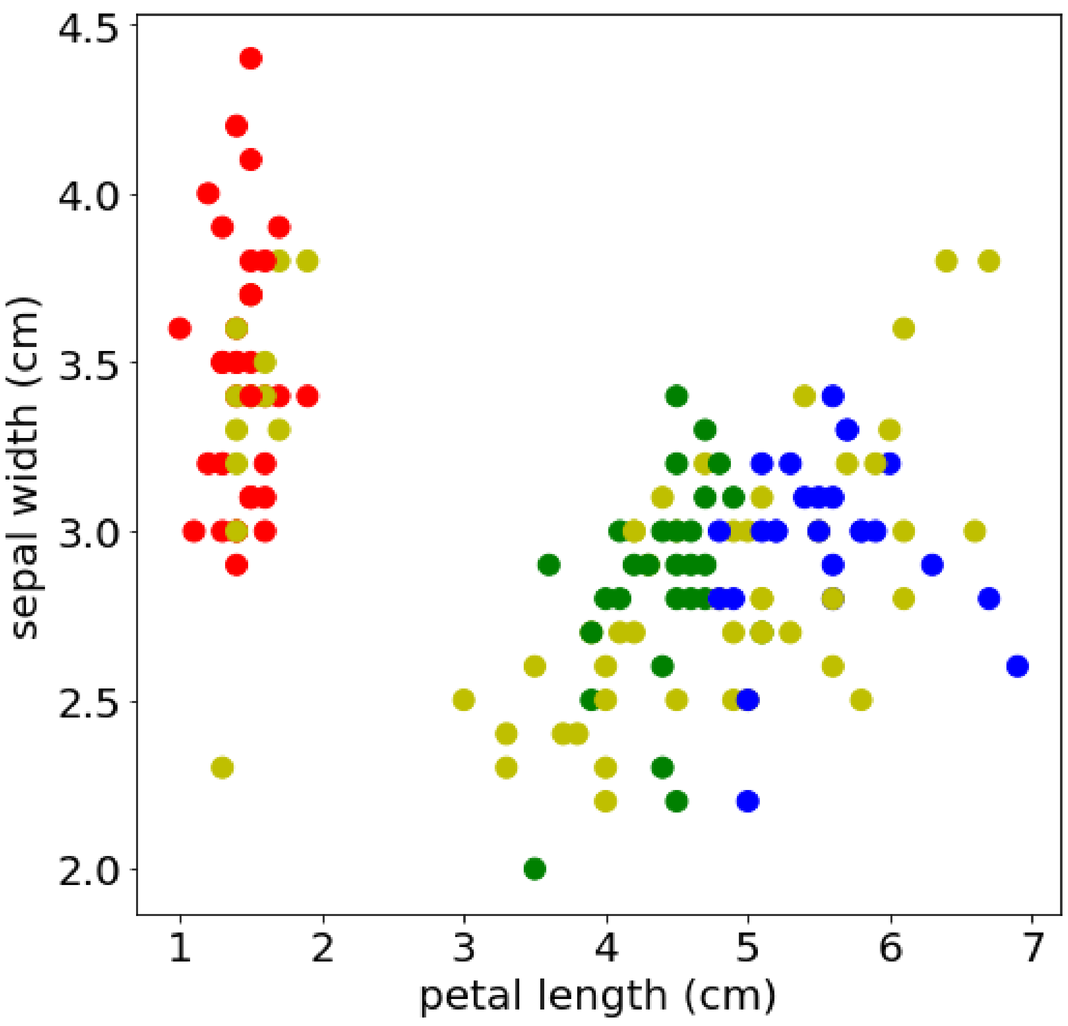
We have to thing about outlierness as a continuous property and our decision is to put a threshold to what we consider far enough from the expected behavior of the data.

Missing Value Imputation

For these experiments we will generate a corrupted copy of the iris dataset by adding some missing values (exactly 75 missing values distributed on the four dimensions) so we can test the effectiveness of the method given the real values.

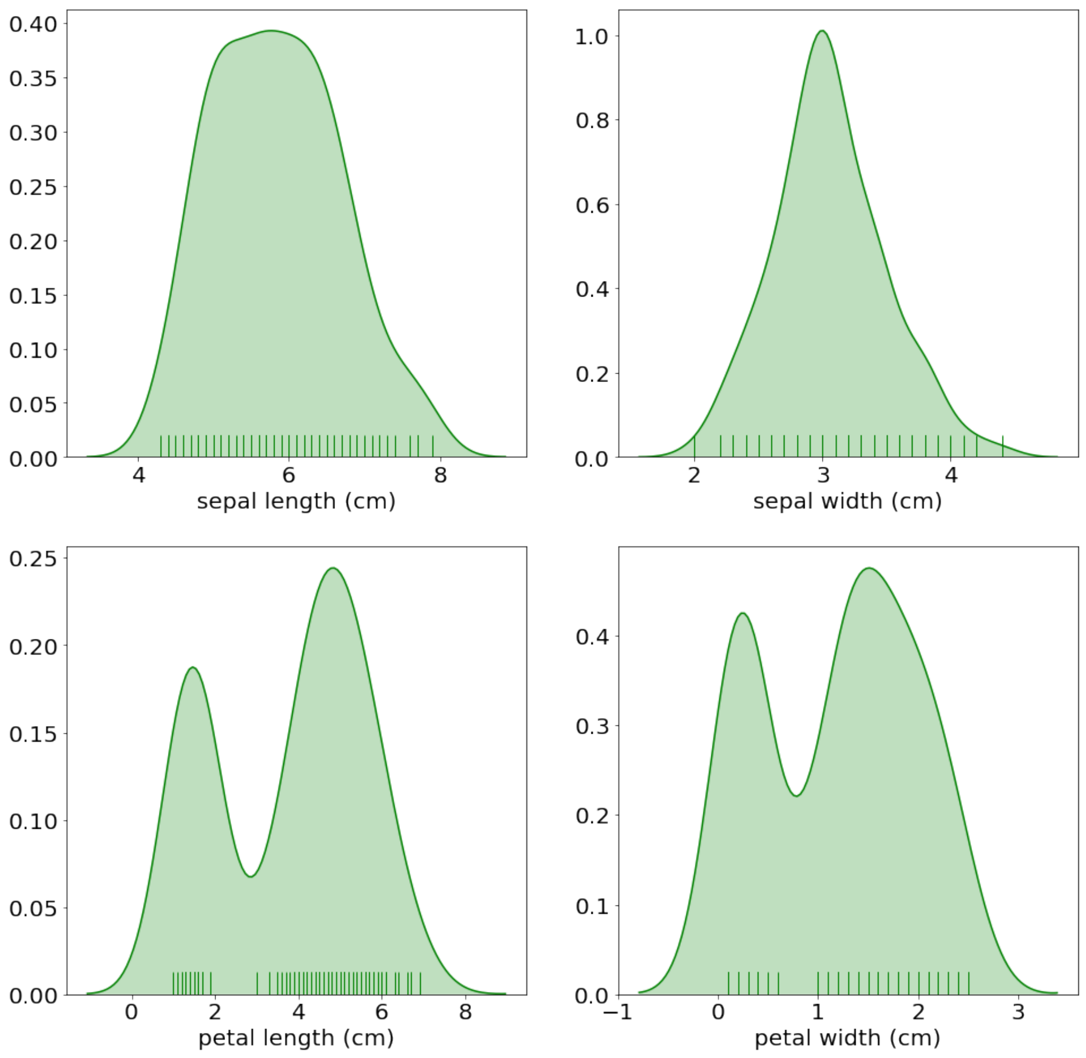
This following plot shows the original data highlighting in yellow the examples that are going to be corrupted.

[9]:
```python
from sklearn.preprocessing import Imputer
from numpy.random import randint
iris = datasets.load_iris()
dimX, dimY = iris['data'].shape
lrandX = randint(dimX, size=75)
lrandY = randint(dimY, size=75)
lcols = [['r','g','b'][i]  for i in iris['target']]
for i in lrandX:
    lcols[i] = 'y'
fig = plt.figure(figsize=(8,8))
plt.xlabel(iris.feature_names[2])
plt.ylabel(iris.feature_names[1])
plt.scatter(iris['data'][:, 2], iris['data'][:, 1],
            c=lcols,s=100);
```



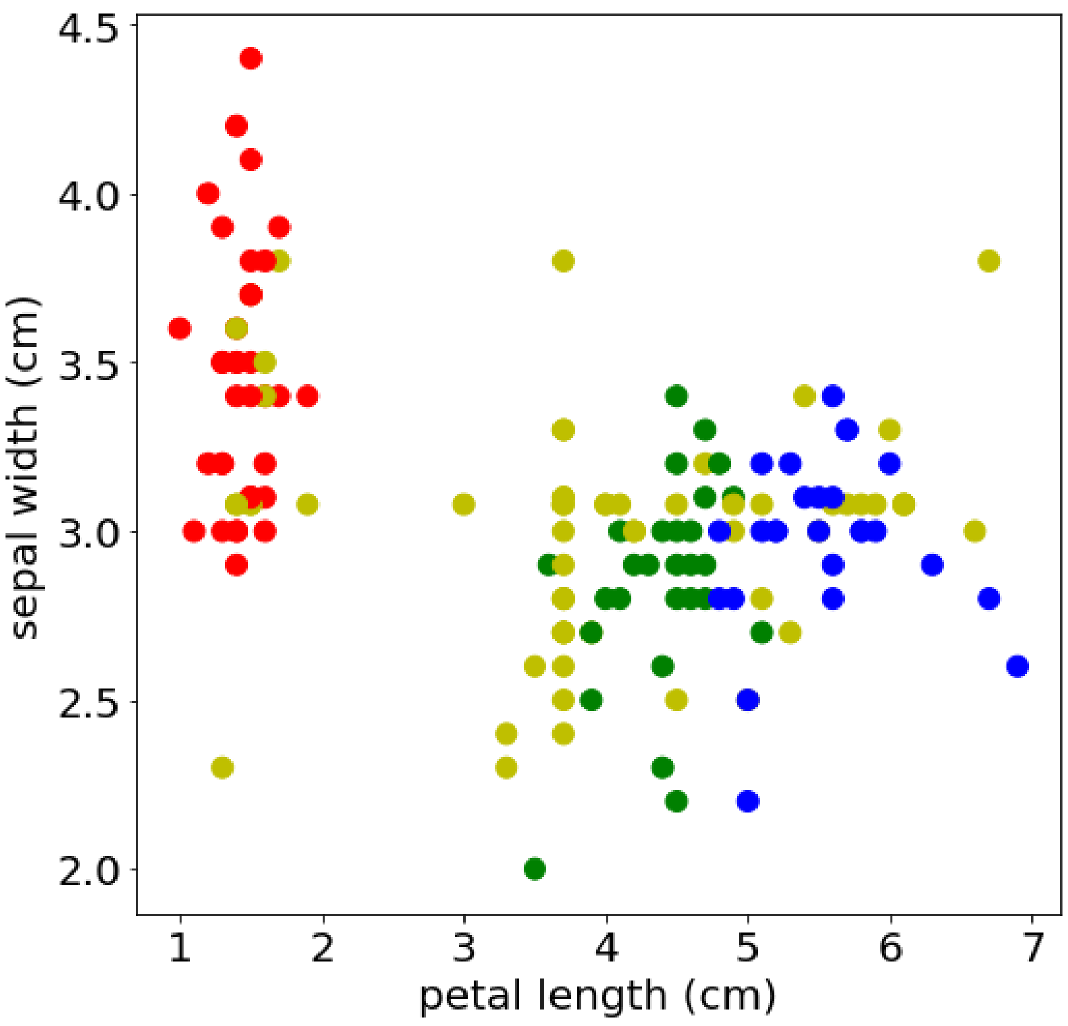
This is a kernel density estimation of the distribution of the values for all the dimensions for the original data (no missing values). We can see that there is a variety of data distributions, with unimodal distributions with smaller and larger variance and bimodal distributions.

[10]:
```python
orig = datasets.load_iris()['data']

fig = plt.figure(figsize=(16,16))
for vshow in range(4):
    ax = fig.add_subplot(2,2,vshow+1)
    plt.xlabel(iris.feature_names[vshow])
    sns.distplot(orig[:,vshow], hist=False, rug=True,
                 color="g", kde_kws={"shade": True})
plt.show()
```



Now we corrupt the data and we apply a missing values imputation algorithm to complete the data, in this case we substitute the missings using the mean of the attribute.

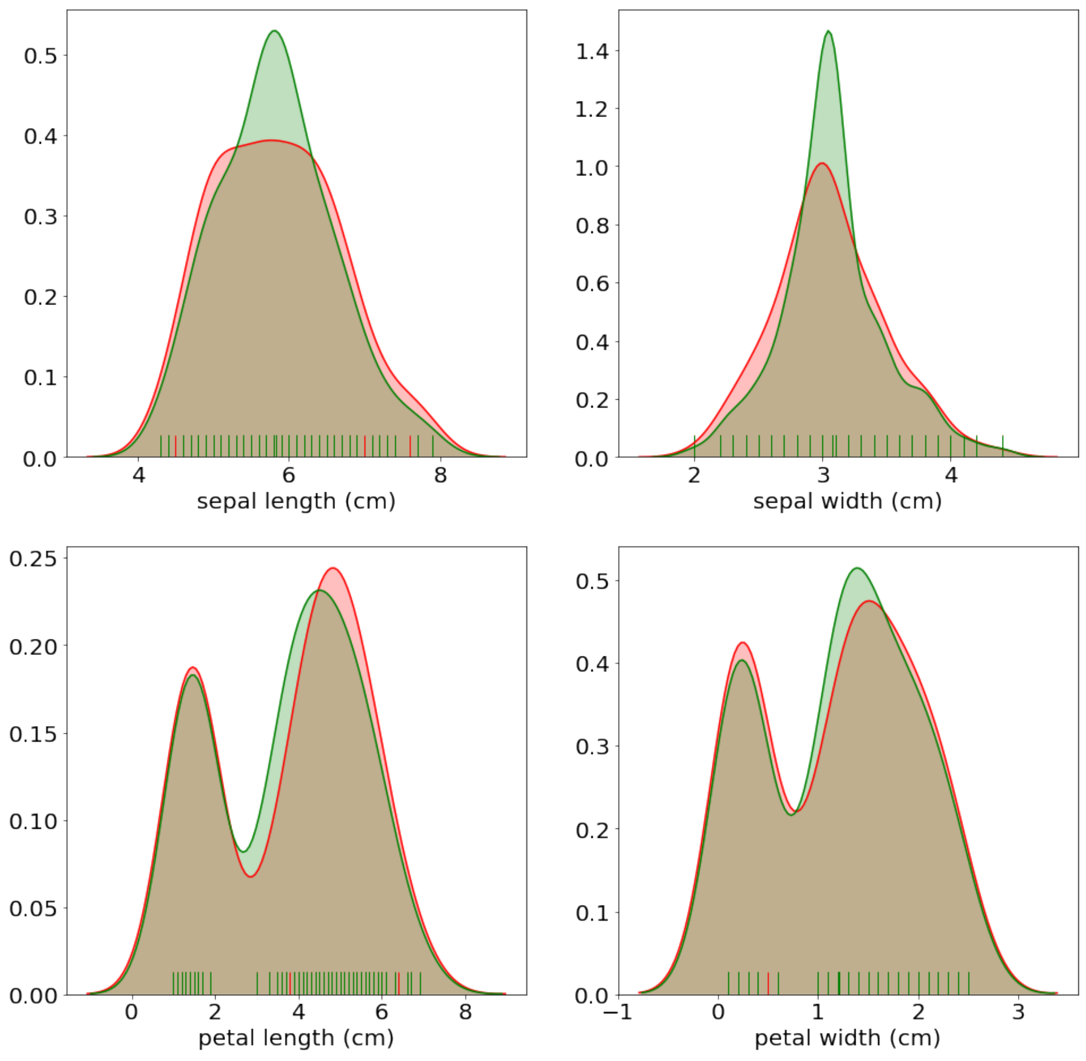
```
[11]: for x,y in zip(lrandX,lrandY):
          iris['data'][x,y]=float('NaN')
      imp = Imputer(missing_values='NaN', strategy='mean')
      imp_iris1 = imp.fit_transform(iris['data'])
      fig = plt.figure(figsize=(8,8))
      plt.xlabel(iris.feature_names[2])
      plt.ylabel(iris.feature_names[1])
      plt.scatter(imp_iris1[:, 2], imp_iris1[:, 1], c=lcols,s=100);
```



As we can see, all the examples with missing values for the dimensions 1 and 2 (the dimensions used in the plot) appear aligned on the mean of the attributes.

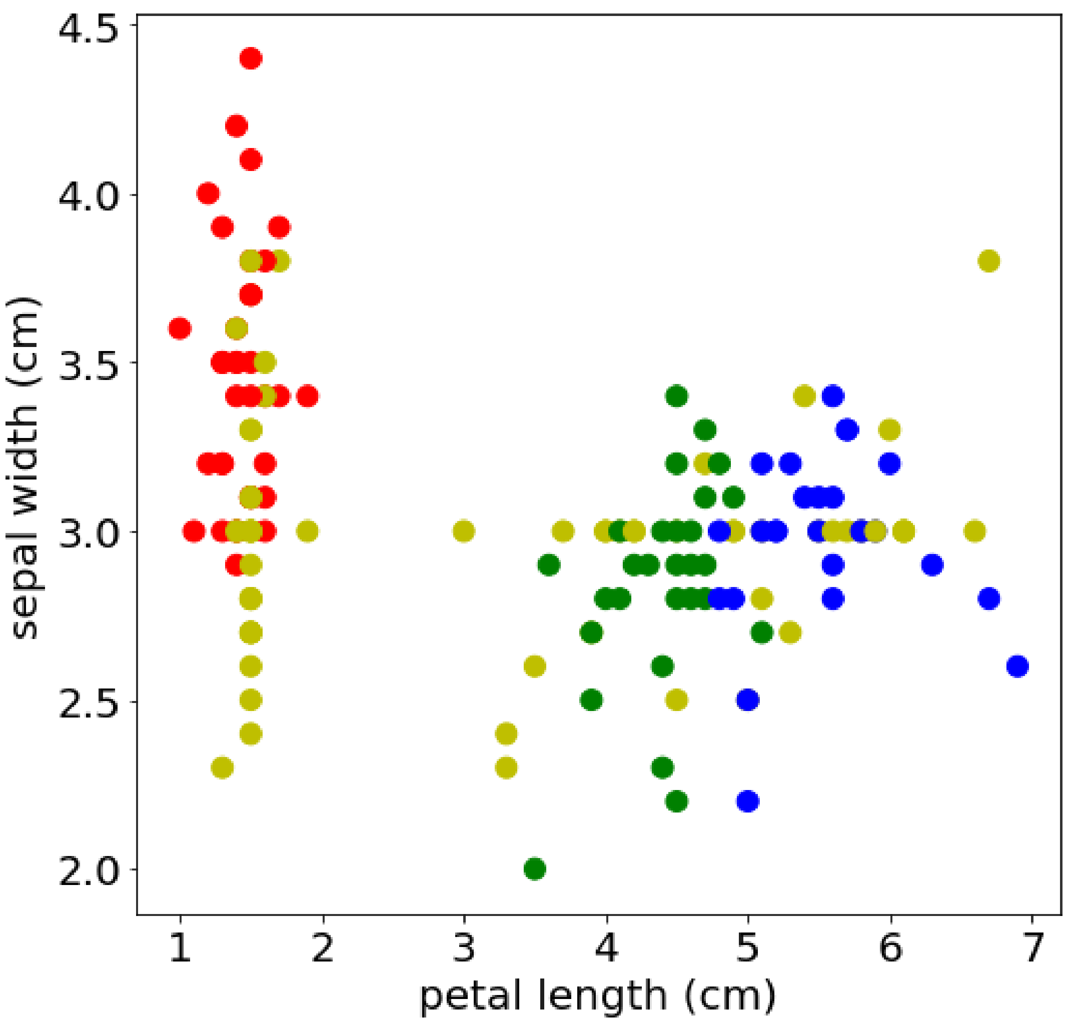
The following plots shows the difference among the original distribution (red) and the distribution after the imputation (green). It can be seen that the distribution of all the dimensions have changed, the variance has been reduce and in the last two variables the mean of the second modality in the joint distribution has changed.

[12]:
```
fig = plt.figure(figsize=(16,16))
for vshow in range(4):
    ax = fig.add_subplot(2,2,vshow+1)
    plt.xlabel(iris.feature_names[vshow])
    sns.distplot(orig[:,vshow], hist=False, rug=True,
                 color="r", kde_kws={"shade": True})
    sns.distplot(imp_iris1[:,vshow], hist=False, rug=True,
                 color="g", kde_kws={"shade": True});
```



Now we use the most frequent value of the attribute to impute the missing values

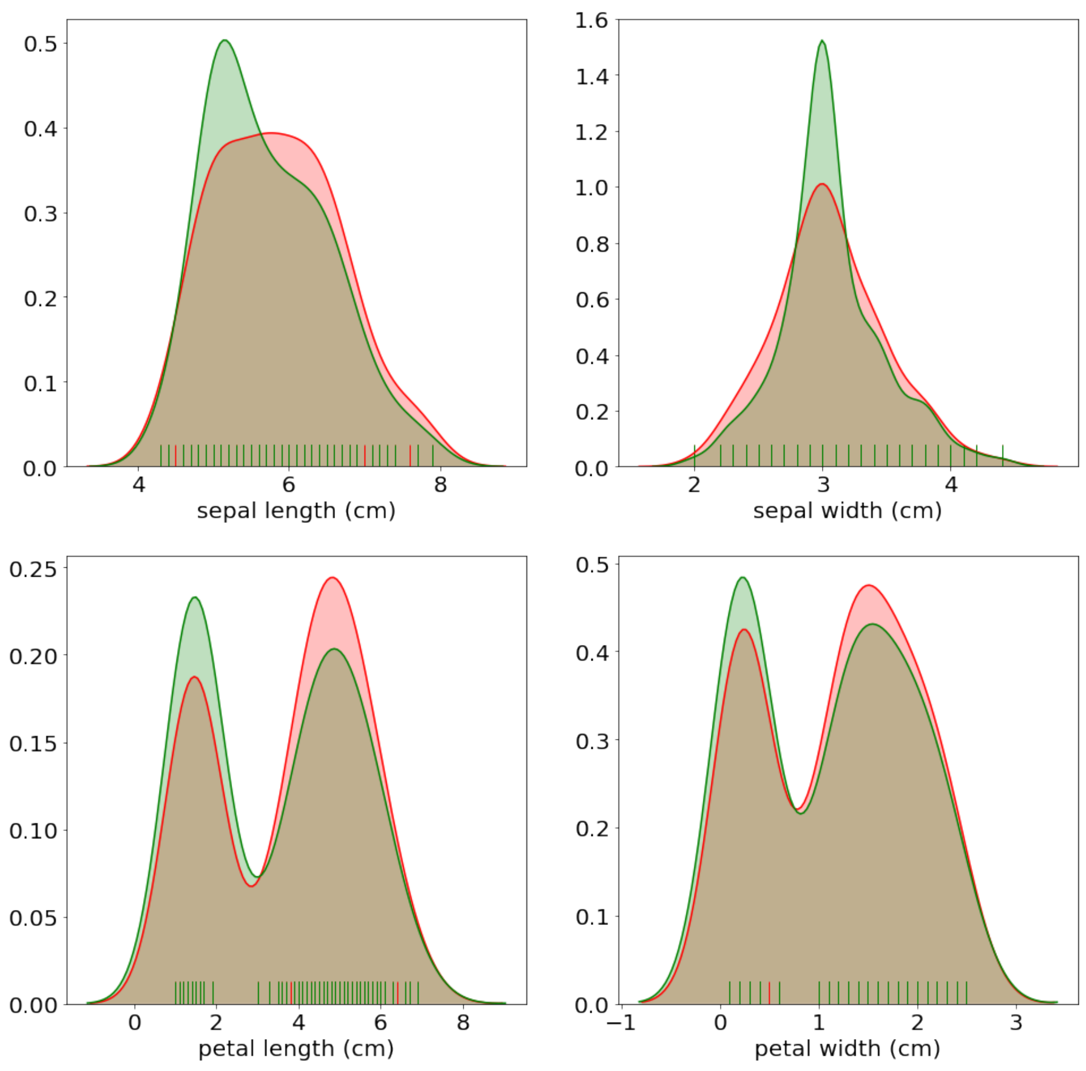
```
[13]: imp = Imputer(missing_values='NaN', strategy='most_frequent')
imp_iris2 = imp.fit_transform(iris['data'])
fig = plt.figure(figsize=(8,8))
plt.xlabel(iris.feature_names[2])
plt.ylabel(iris.feature_names[1])
plt.scatter(imp_iris2[:, 2], imp_iris2[:, 1],
            c=lcols,s=100);
```



As expected, the imputed examples now appear aligned on the most frequent value and the variance is also reduced.

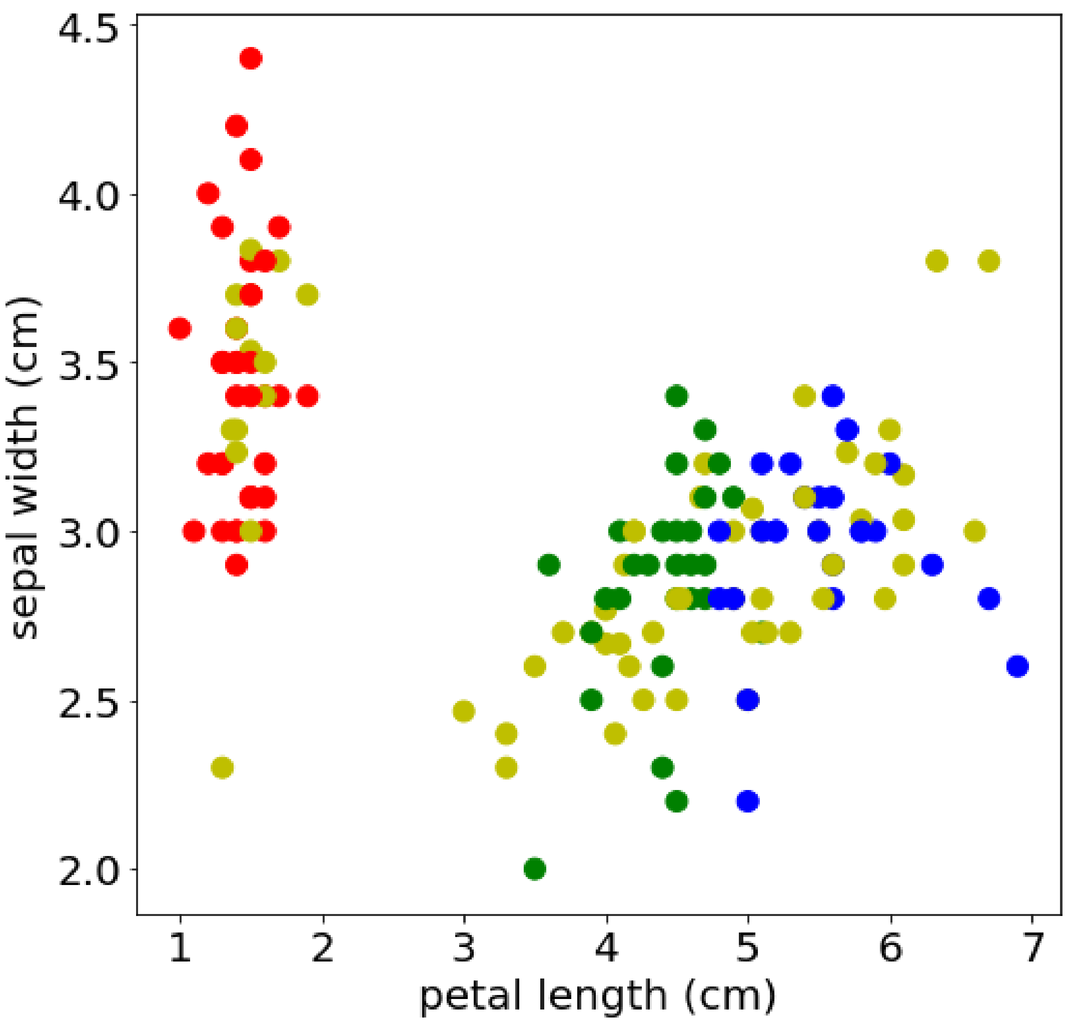
All the new densities have changed and in some variables examples from one density have been transfered to the other.

```
[14]: fig = plt.figure(figsize=(16,16))
      for vshow in range(4):
          ax = fig.add_subplot(2,2,vshow+1)
          plt.xlabel(iris.feature_names[vshow])
          sns.distplot(orig[:,vshow], hist=False, rug=True,
                       color="r", kde_kws={"shade": True})
          sns.distplot(imp_iris2[:,vshow], hist=False, rug=True,
                       color="g", kde_kws={"shade": True});
```



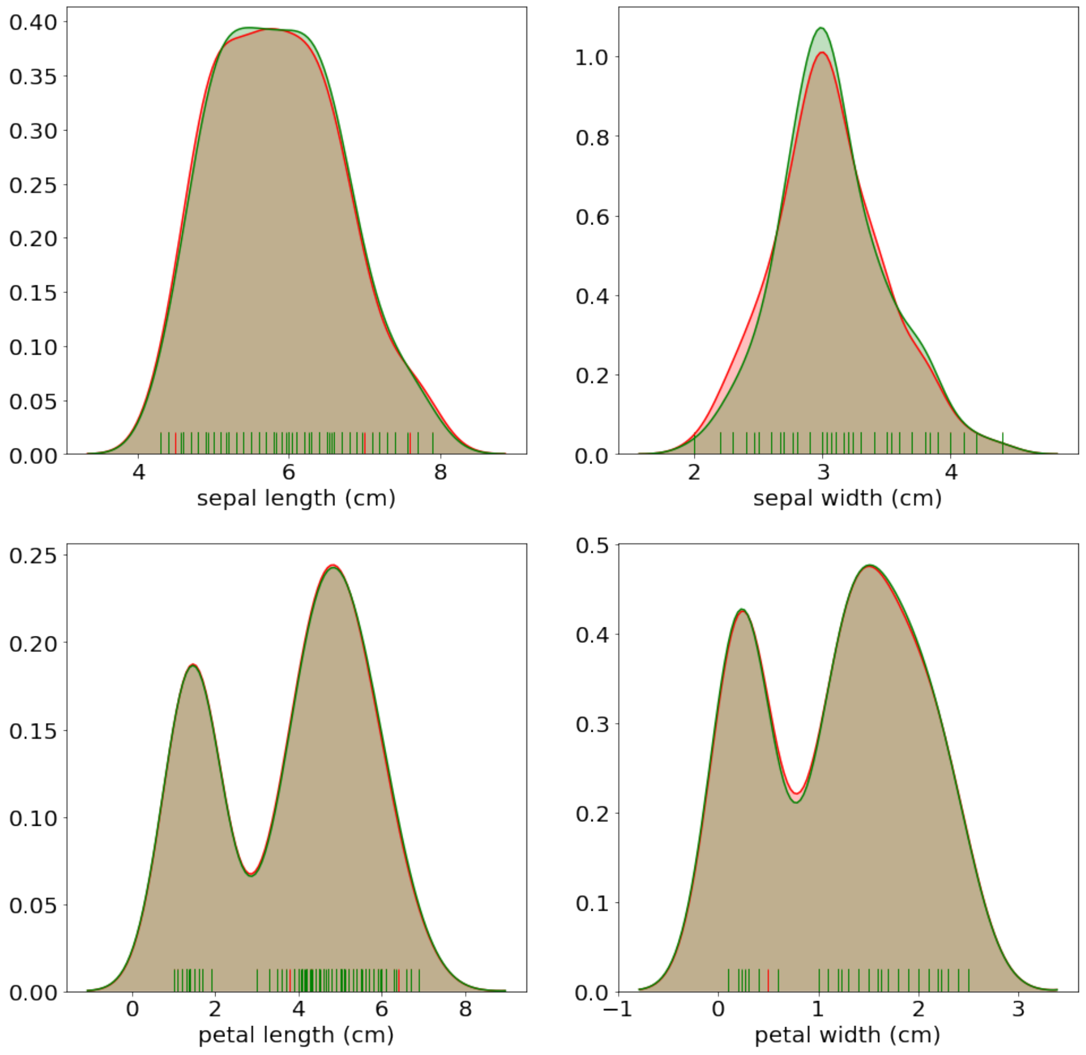
Now we are going to use the euclidean distance to determine the closest examples and to use the mean of the values of the 3-nearest neighbor to substitute the missing value

```
[15]: from kemlglearn.preprocessing import KnnImputer
knnimp = KnnImputer(missing_values='NaN', n_neighbors=3)
imp_iris3 = knnimp.fit_transform(iris['data'])
fig = plt.figure(figsize=(8,8))
plt.xlabel(iris.feature_names[2])
plt.ylabel(iris.feature_names[1])
plt.scatter(imp_iris3[:, 2], imp_iris3[:, 1],
            c=lcols,s=100);
```



As we can see the examples look more naturally distributed and now the distribution of the attributes looks more similar to the original one.

```
[16]: fig = plt.figure(figsize=(16,16))
      for vshow in range(4):
          ax = fig.add_subplot(2,2,vshow+1)
          plt.xlabel(iris.feature_names[vshow])
          sns.distplot(orig[:,vshow], hist=False, rug=True,
                       color="r", kde_kws={"shade": True})
          sns.distplot(imp_iris3[:,vshow], hist=False, rug=True,
                       color="g", kde_kws={"shade": True});
```



1.2.4 Data normalization

Data normalization is applied to quantitative attributes in order to eliminate the effects of having different scale measures. To have disparate scales in the attributes makes difficult to compare the data and affects the measure of similarity or distance among examples. This normalization can be explicit, a new dataset is generated, or implicit, the data is normalized when it is processed by the algorithm that is used.

Range normalization

Also known as min-max scaling, it transforms all the values of the attributes to a predetermined scale (for instance $[0,1]$ or $[-1,1]$). This can be computed from the range of values of an attribute, the following transformation changes the values of a variable to the range $[0,1]$

$$range\_norm(x) = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{1.3}$$

From this normalization other ranges can be obtained simply by translation and scaling.

Standard score normalization or Z-normalization:

It transforms the data assuming gaussian distribution, using the attributes empirical moments (μ_x, σ_x) so all the attributes have $\mu = 0$ and $\sigma = 1$. The following transformation performs this change:

$$Z\_norm(x) = \frac{x - \mu_x}{\sigma_x} \tag{1.4}$$

A variation of this scaling is to use the *mean absolute deviation* instead of the standard deviation. This reduces to an extent the effect of the outliers in the dataset. The mean absolute deviation is computed as:

$$s_x = \frac{1}{N}\sum_{i=0}^{N} |x - \mu_x| \tag{1.5}$$

Robust Scaling

One of the problems of the previous methods is that they do not have in account possible outliers in the data. If they have not been removed the resulting scaled data can be very skewed and not directly comparable with other attributes that not present outliers. This can be corrected using methods from robust statistics, that do not compute transformation using the usual estimators for the mean and the deviation of the data. One robust technique is to use for transformation the median and the interquartile range (IRQ). The IRQ is usually computed as the spread of the middle 50% percent of the data, the 1st quartile (25%) and the 3rd quartile (75%)..

Quantile and power transformations

Sometimes we are also interested on transforming the data to a more suitable distribution for the task we want to solve, for instance, to uniform or gaussian distributions. This can also be combined with scaling.

The quantile transformation is a non-parametric non-linear transformation method that uses the information of the quantiles computed from the data to match it to the target distribution. The quantiles are a discretization of the cumulative probability distribution function, and we can use it to transform each quantile to the theoretical quantile distribution of a uniform or gaussian distributions.

Power transformations are parametric non-linear transformations that match the original distribution of the data with a distribution that is closer to gaussianity. There are different methods like the Box-Cox, transformation that only can be applied to positive data, and the Yeo-Johnson transformation that can be applied to positive and negative data.

The Box-Cox transformations finds the best λ parameter (closest to gaussian) for the transformation:

$$x(\lambda) = \begin{cases} \frac{x^{\lambda}-1}{\lambda} & \lambda \neq 0 \\ \log(x) + \sigma^2 & \lambda = 0 \end{cases} \tag{1.6}$$

The Yeo-Johnson transformation finds the best λ parameter (closest to gaussian) for the transformation:

$$x(\lambda) = \begin{cases} \frac{(x+1)^{\lambda}-1}{\lambda} & \lambda \neq 0, x \geq 0 \\ \log(x+1) + \sigma^2 & \lambda = 0, x \geq 0 \\ -\frac{(x+1)^{2-\lambda}-1}{2-\lambda} & \lambda \neq 2, x < 0 \\ -\log(-x+1) + \sigma^2 & \lambda = 0, x < 0 \end{cases} \tag{1.7}$$

Obviously there is not guarantee that the best match follows a strictly a gaussian distribution, so it has to be checked (at least visually) before being used.

1.2.5 Data discretization

Sometimes the process that we are going to apply to the data does not admit continuous values, or it is more simple to understand the results if the values are discrete. Data discretization allows transforming attribute values from quantitative to qualitative. There are different ways of performing this task, we are only going to describe the unsupervised ones.

The task is to decide how to partition the set of continuous values in a set of bins computing the limits between consecutive bins. The number of bins is usually fixed beforehand using domain information, or can be adjusted using some quality measures.

- **Equal length bins:** The range between the maximum and the minimum value is divided in as many intervals of equal length as the desired bins. Values are discretized accordingly to the interval they belong to.
- **Equal frequency bins:** The intervals are computed so each bin has the same number of examples, the length of the intervals will be consequently different. The cutting points are usually computed as the mean of the examples that are in the extremes of consecutive bins.
- **Distribution approximation:** A histogram of the data can be computed (using a specific number of bins), and a parameterized function is fitted to approximate the frequencies (e.g. a polynomial). The intervals are selected computing the different minimums of the function. Alternatively, Kernel Density Estimation (non parametric) can be applied to approximate the different densities of the data, and the intervals can be determined by finding the areas with lower density.

These three methods are represented in figure 11.7. Several other techniques can be applied to unsupervisedly detect the different modalities of the data, like for example entropy based measures, Minimum Description Length or even clustering.

As an example, figure 1.7 shows the effect of equal width and equal frequency discretization using five bins for the well known Iris dataset. It can be observed the different distribution of the examples on the bins for each one of the discretizations, jittering has been used in the representation, so the amount of examples in each combination can be appreciated.

Supervised discretization methods use class information to decide the best way to split the data, and the optimal number of bins. Unfortunately, this information is not always available.

1.2.6 Data Preprocessing: Discretization and Normalization

This notebook shows some of the discretization and normalization techniques explained in the course.

[1]:
```python
from sklearn import datasets
from sklearn.preprocessing import KBinsDiscretizer
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import matplotlib
warnings.filterwarnings('ignore')
matplotlib.rcParams.update({'font.size':16})

iris = datasets.load_iris()
col = ['r', 'g', 'b']
lc = [col[i] for i in iris['target']]
```

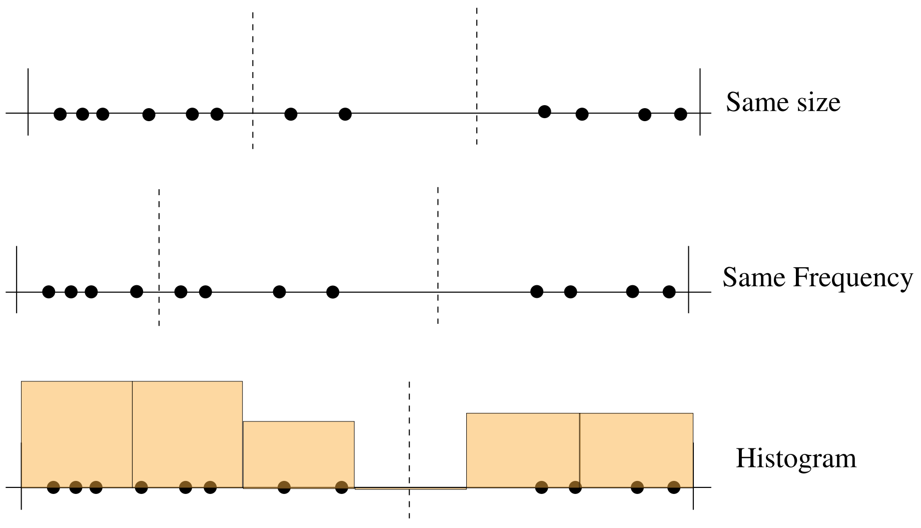


Figure 1.6: Attribute discretization methods

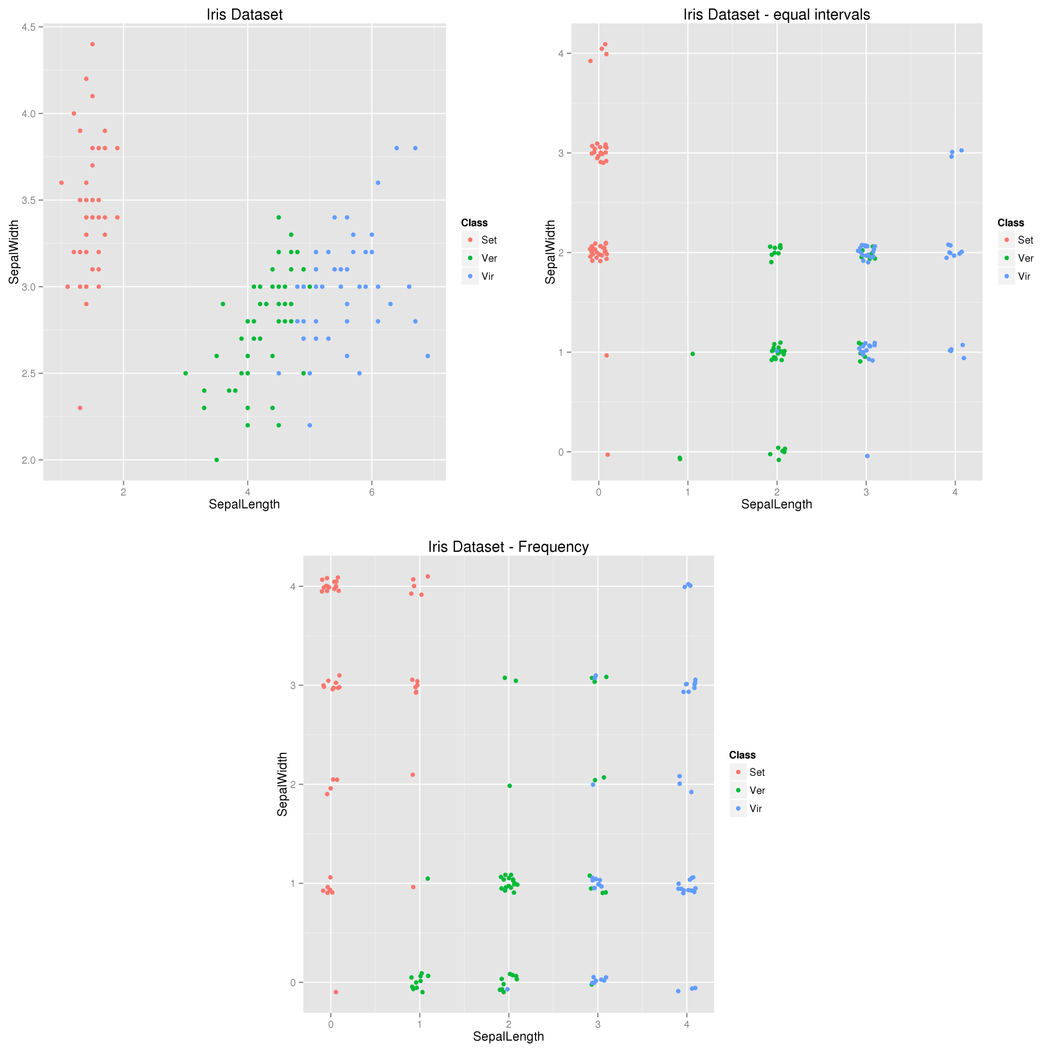
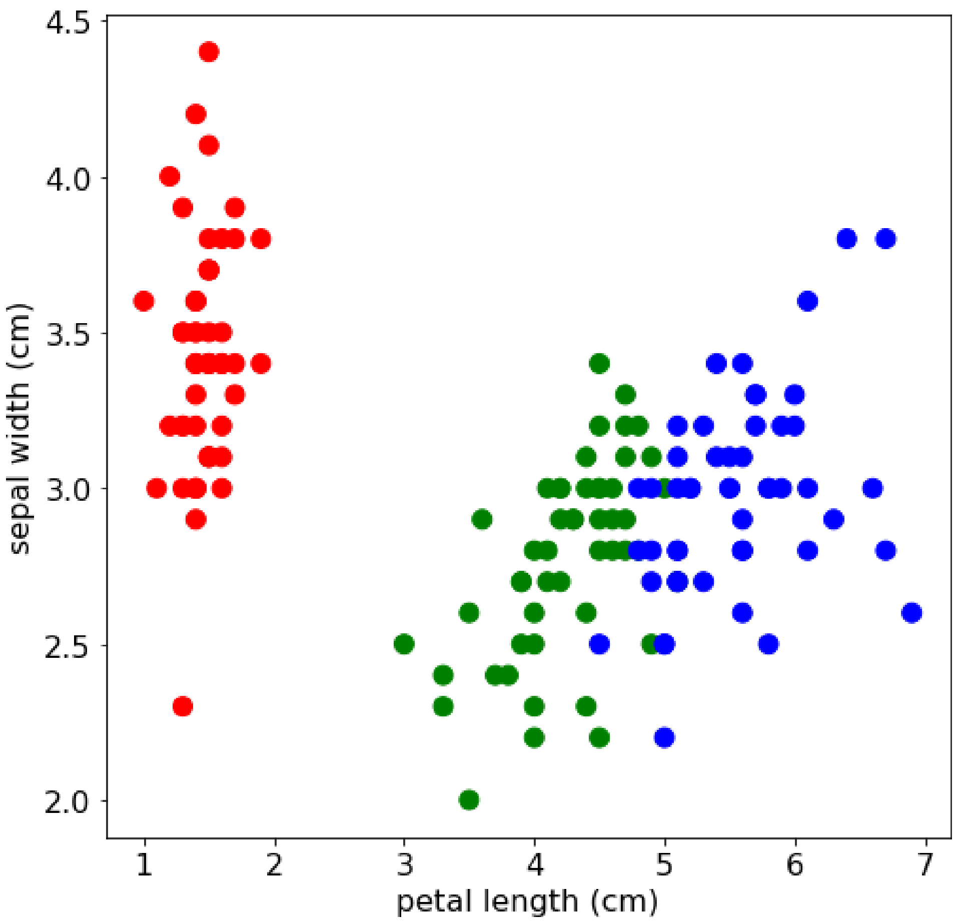


Figure 1.7: Equal sized and frequency discretization applied to the iris dataset

Attributes Discretization

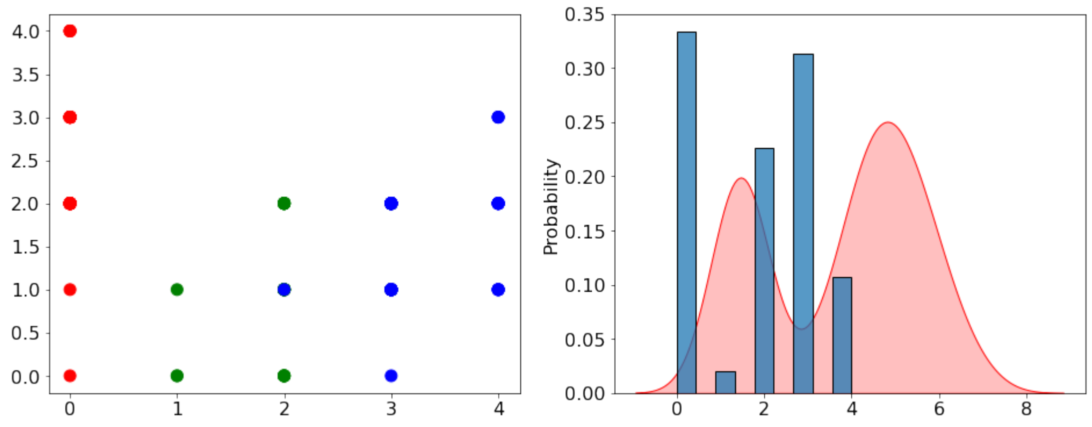
We will use the iris dataset as exaple, the following plot corresponds to two of the attributes of the Iris dataset. These are continuous values that can be discretize in different ways to a range of values.

[2]:
```
fig = plt.figure(figsize=(8,8))
plt.xlabel(iris.feature_names[2])
plt.ylabel(iris.feature_names[1])
plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=lc,s=100);
```



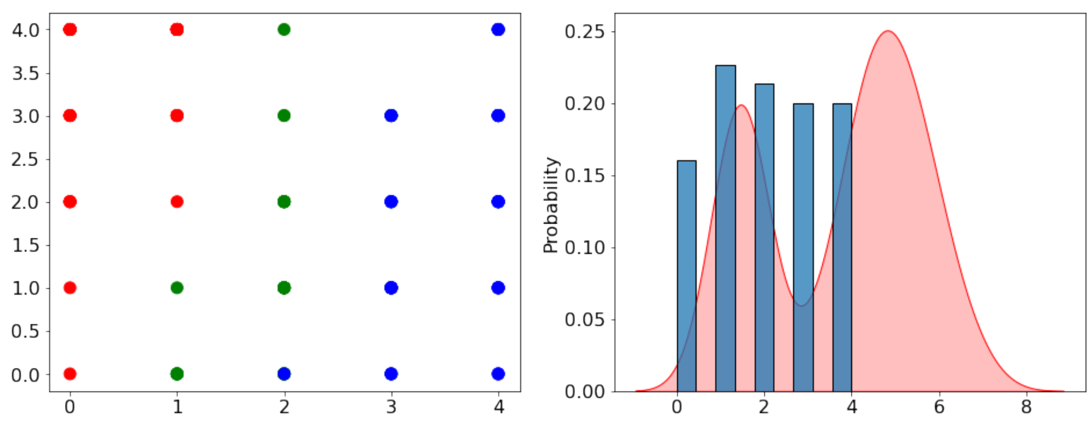
We can begin discretizing the attributes using equal bins (5 in this examples), the plot represents the 5x5 possible combinations, one combination can have many data points. The number of bins is arbitrary and should be djusted depending on the knowlege about the data or the specific application.

[3]:
```
bins=5

disc = KBinsDiscretizer(n_bins=bins, encode='ordinal', strategy='uniform')
disc.fit(iris['data'])
irisdisc = disc.transform(iris['data'])
fig = plt.figure(figsize=(16,6))
plt.subplot(1,2,1)
plt.scatter(irisdisc[:, 2], irisdisc[:, 1], c=lc,s=100)
plt.subplot(1,2,2)
sns.histplot(irisdisc[:, 2],stat='probability')
sns.kdeplot(x=iris['data'][:,2], shade=True,  color="r");
```



Using frequency distretization obtains also a 5x5 grid of combinations but the distribution of the data points changes. This should be more faithful to the distribution of the data along the dimension, so sections with less probability will correspond to wider ranges and sections with more probability.

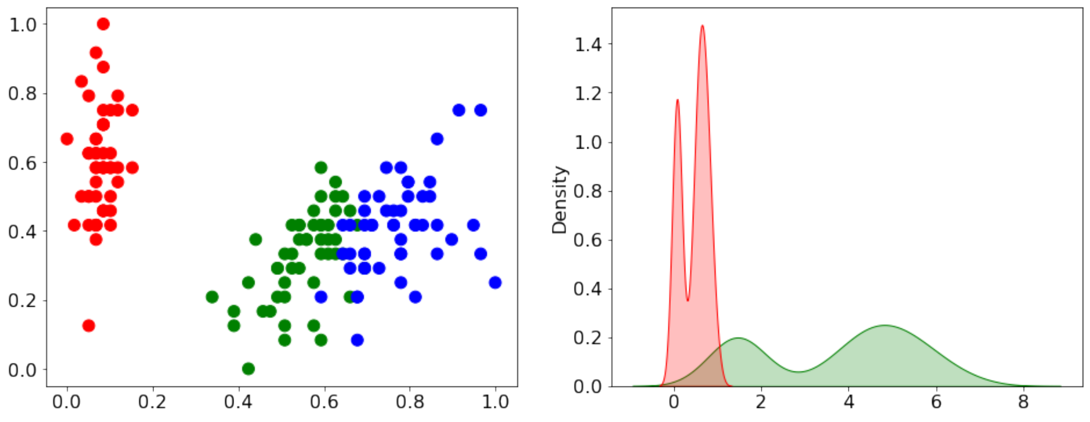
[4]:
```
bins=5

disc = KBinsDiscretizer(n_bins=bins, encode='ordinal', strategy='quantile')
disc.fit(iris['data'])
irisdisc = disc.transform(iris['data'])
fig = plt.figure(figsize=(16,6))
plt.subplot(1,2,1)
plt.scatter(irisdisc[:, 2], irisdisc[:, 1], c=lc,s=100)
plt.subplot(1,2,2)
sns.histplot(irisdisc[:, 2],stat='probability')
sns.kdeplot(x=iris['data'][:,2], shade=True,  color="r");
```



Data normalization

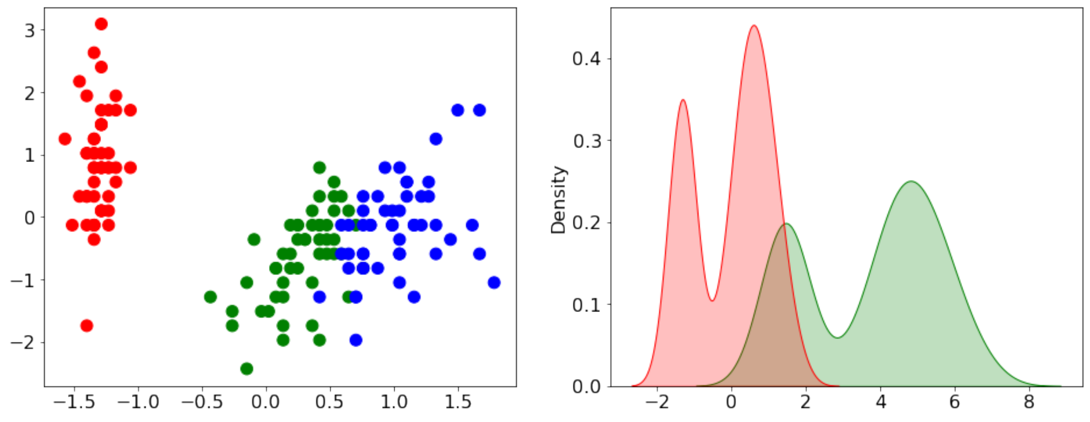
Data normalization standardizes the attributes so their scales do not influence the comparisons, first we will use a range normalization for the iris dataset, the only difference that we can observe in the plot is the change of scale (plots already scale the ranges of the attributes to maintain a 1:1 proportion), but we can see the difference compared with the original data.

[5]:
```python
from sklearn.preprocessing import StandardScaler, MinMaxScaler,␣
 ↪PowerTransformer, QuantileTransformer
import seaborn as sns

mx = MinMaxScaler()
fdata = mx.fit_transform(iris['data'])
fig = plt.figure(figsize=(16,6))
plt.subplot(1,2,1)
plt.scatter(fdata[:, 2], fdata[:, 1], c=lc,s=100);
plt.subplot(1,2,2)
sns.kdeplot(x=iris['data'][:,2], shade=True, color="g")
sns.kdeplot(fdata[:,2] , shade=True,  color="r");
```



We can now use a standard score normalization assuming gaussian data, as before the only change that we can observe is the change in the scale of the axis of the plot, but we can see the difference compared with the original data.

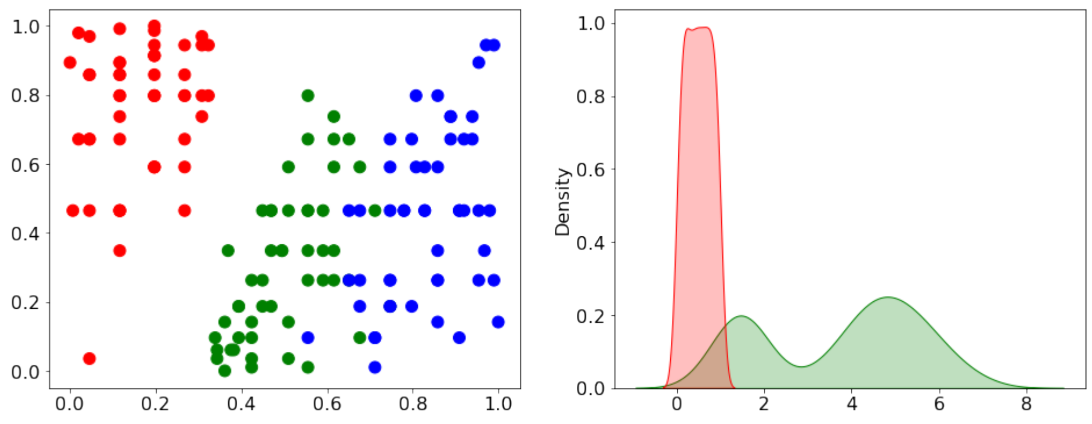
[6]:
```python
std = StandardScaler()
fdata = std.fit_transform(iris['data'])
fig = plt.figure(figsize=(16,6))
plt.subplot(1,2,1)
plt.scatter(fdata[:, 2], fdata[:, 1], c=lc,s=100);
plt.subplot(1,2,2)
sns.kdeplot(x=iris['data'][:,2], shade=True, color="g")
sns.kdeplot(fdata[:,2] , shade=True,  color="r");
```



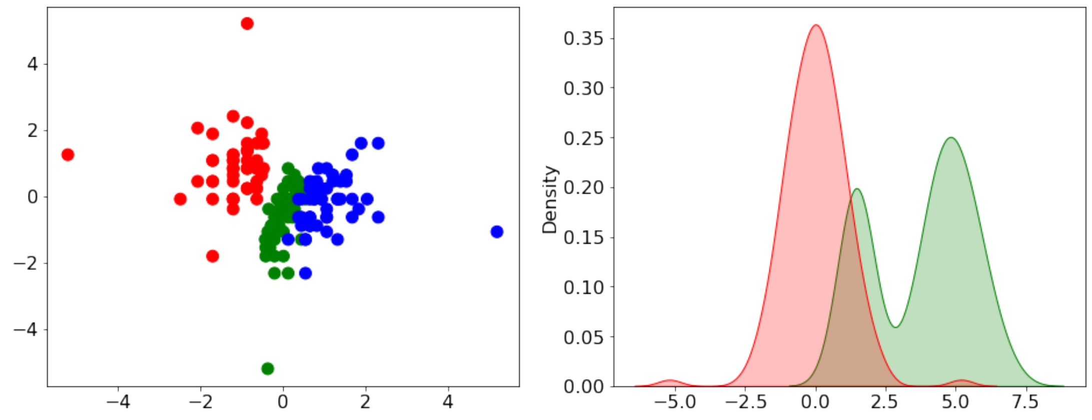
With the quantile transformation we can obtain a “normal” or “uniform” distribution by mapping the CDF of the original data to the CDF of the targer distribution, this can distort the data, but it can be useful when the methods we will use need the data to have a specific distribution.

Notice that we can perform the mapping even when the distribution is complex.

[7]:
```
qt = QuantileTransformer(n_quantiles=100, output_distribution='uniform')
fdata = qt.fit_transform(iris['data'])
fig = plt.figure(figsize=(16,6))
plt.subplot(1,2,1)
plt.scatter(fdata[:, 2], fdata[:, 1], c=lc,s=100);
plt.subplot(1,2,2)
sns.kdeplot(x=iris['data'][:,2], shade=True, color="g")
sns.kdeplot(fdata[:,2] , shade=True,  color="r");
```

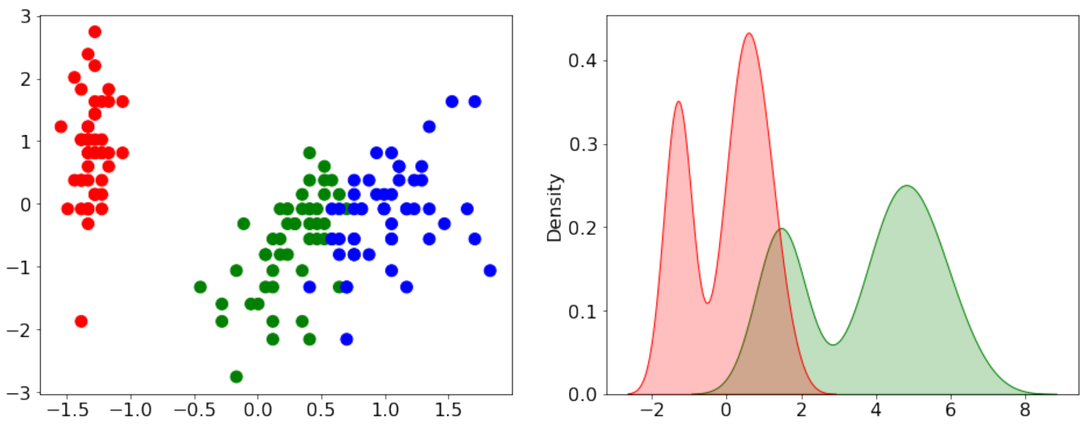


[8]:
```
qt = QuantileTransformer(n_quantiles=100, output_distribution='normal')
fdata = qt.fit_transform(iris['data'])
fig = plt.figure(figsize=(16,6))
plt.subplot(1,2,1)
plt.scatter(fdata[:, 2], fdata[:, 1], c=lc,s=100);
plt.subplot(1,2,2)
sns.kdeplot(x=iris['data'][:,2], shade=True, color="g")
sns.kdeplot(fdata[:,2] , shade=True,  color="r");
```



The power transformation applies a parameterized function to obtain a more gaussian distribution if possible

[9]:
```
pt = PowerTransformer()
fdata = pt.fit_transform(iris['data'])
fig = plt.figure(figsize=(16,6))
plt.subplot(1,2,1)
plt.scatter(fdata[:, 2], fdata[:, 1], c=lc,s=100);
plt.subplot(1,2,2)
sns.kdeplot(x=iris['data'][:,2], shade=True, color="g")
sns.kdeplot(fdata[:,2] , shade=True,  color="r");
```



1.3 Dimensionality reduction

Before applying a specific mining task on a dataset, several preprocessing steps can be performed. The first goal of the preprocessing step is to assure the quality of the data by reducing the noisy and irrelevant information that it could contain, this can be achieved by using methods described on the previous sections. The second goal is to reduce the size of the dataset, so the computational cost of the discovery task is also reduced. This reduction process can also have the purpose of visualizing the data to help the discovery process.

There are two dimensions that have to be taken into account when reducing the size of a dataset. The first one is the number of examples. This problem can be addressed by sampling techniques when it is clear that a smaller subset of the data holds the same information that the whole dataset. Not for all applications this is the case, and sometimes the specific goal of the mining process is to find particular groups of examples with low frequency, but of high value. These data could be discarded by the sampling process, making unfruitful the process. In other applications, the data is a stream, this circumstance makes more difficult the sampling process or carries the risk of losing important information from the data if its distribution changes over time.

Dimensionality reduction addresses the number of attributes of the dataset by transforming it from its original representation to one with a reduced set of features. These techniques come from very different areas, but the common goal is to obtain a new dataset that preserves, up to a level, the original structure of the data, so its analysis will result in the same or equivalent patterns present in the original data.

Broadly, there are two kinds of methods for reducing the attributes in a dataset, feature selection and feature extraction.

1.3.1 Feature selection

Most of the research on feature selection is related to supervised learning [82]. More recently, methods for unsupervised learning have appeared in the literature [27], [61], [137], [140]. These methods can be divided on *filters*, that use characteristics of the features to determine their salience so the more relevant ones can be kept, and *wrappers*, that involve the exploration of the space of subsets of features and the use of a clustering algorithm to evaluate the quality of the partitions generated with each subset, according to an internal or external quality criteria. The main advantage of all these methods is that they preserve the original attributes, so the resulting patterns can be interpreted more easily.

Wrapper methods need a model to evaluate the relevance of subsets of attributes. This is easier for supervised learning because the labels for the examples are available. The success of the methods depend on the chosen model and how well it captures the inherent structure of the data. Typical unsupervised wrapper methods involve the use of a clustering algorithm that computes, as part of the fitting of the model, a set of weights for each attribute or the use of an objective function that penalizes the size of the model.

Filter methods use measures that assess the relevance of each attribute. This is also easier for supervised methods, because they can measure directly the link between each attribute and the labels. For the unsupervised case, the idea is to obtain a measure that evaluates the capacity of each attribute to reveal the structure of the data (class separability, similarity of instances in the same class). It is typical to use measures for properties about the spatial structure of the data (Entropy, PCA based scores, laplacian matrix based scores) or measures of feature correlation/dependence.

Laplacian score

The Laplacian Score [61] is a filter method that ranks the features respect to their ability of preserving the natural structure of the data. It is assumed that there are compact clusters in the data and this will reflect on the local relations among the examples.

In order to measure these local relations, this method uses the spectral matrix of the graph computed from the near neighbors of the examples. The edges of the graph use as weights the similarity among the examples. This means that two parameters have to be decided. The first one is the number of neighbors used to build the graph and, the second one, the similarity measure to be used to compute the weights.

This similarity is usually computed using a gaussian kernel with a specific bandwidth (σ) applied to the distances among examples, so the weights of the edges are:

$$S_{ij} = e^{\frac{||x_i - x_j||^2}{\sigma}} \tag{1.8}$$

And all edges not present have a value of 0.

Being S the similarity matrix and D a diagonal matrix where the elements of the diagonal are the sum of the weights of the row, the Laplacian matrix can be computed as $L = S - D$.

Using the values of an attribute of the dataset f_r and being 1 a vector of ones, the score first computes $\tilde{f}_r$ as:

$$\tilde{f}_r = f_r - \frac{f_r^T D 1}{1^T D 1} 1 \tag{1.9}$$

The score L_r is then computed as:

$$L_r = \frac{\tilde{f}_r^T L \tilde{f}_r}{\tilde{f}_r^T D \tilde{f}_r} \tag{1.10}$$

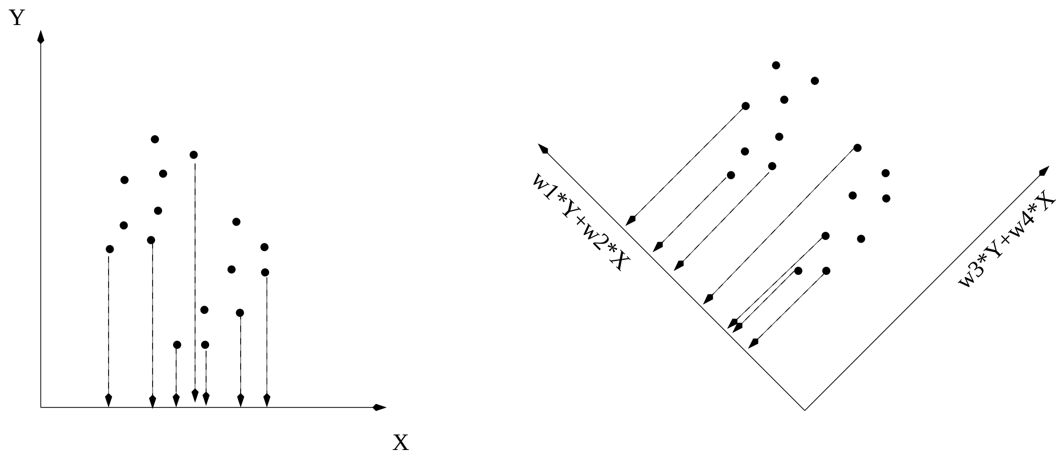


Figure 1.8: PCA computes a transformation that needs less dimensionality to explain the variance of the data

This gives a ranking for the relevance of the attributes and only the k attributes with the higher score are kept. The number k could be a predefined value or can be obtained by looking for a jump in the values of the score for two consecutive attributes in the ranking.

1.3.2 Feature extraction

Feature extraction is an area with many methods. The goal is to build a new set of attributes that maintains some characteristics of the data that also preserve the patterns. New attributes are computed as combinations (linear or not) of the original attributes. The following sections describe the principal methods used in this area, a more extensive tutorial can be found in [16].

Principal Component Analysis (PCA)

The most popular method for feature extraction is Principal Component Analysis [58]. This method computes a linear transformation of the data that results in a set of orthogonal dimensions that account for all the variance of the dataset. It is usual for only a few of these dimensions to hold most of the variance, meaning that with only this subset should be enough to discover the patterns in the data. This also allows visualizing the structure of the data, if two or three dimensions hold a large portion of the variance.

PCA is best suited when the attributes follow a gaussian distribution. The new set of features, called *principal components*, are uncorrelated, so each one explains a part of the total variance.

The method can be described geometrically as a rotation/scaling of the axes looking for the projection where the variance of the data can be explained by a few variables. Figure 1.8 represents the goal of this method. The projections of the data on the original dimensions does not allow discovering the structure of the data, but a linear transformation of the dimensions results in a new dataset where only the projection in one dimension is enough to capture this structure.

The computation of the principal components obtains the best linear approximation of the data

$$f(\lambda) = \mu + V_q\lambda \tag{1.11}$$

where μ is a location vector in $\mathbb{R}^p$, V_q is a $p \times q$ matrix of q orthogonal unit vectors and λ is a

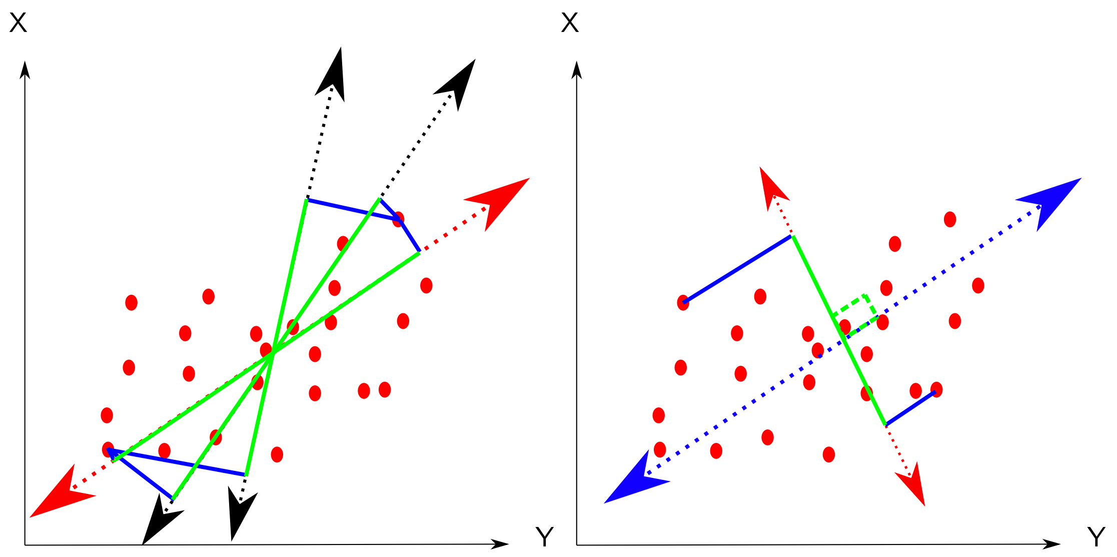


Figure 1.9: Intuitive computation of the PCA

q vector of parameters We want to minimize the reconstruction error for the data:

$$\min_{\mu,\{\lambda_i\},V_q} \sum_{i=1}^{N} ||x_i - \mu - V_q\lambda_i||^2 \tag{1.12}$$

Optimizing partially for μ and λ_i (taking partial derivatives) we have that:

$$\mu = \bar{x} \tag{1.13}$$
$$\lambda_i = V_{iq}^T(x_i - \bar{x}) \tag{1.14}$$

We can obtain the matrix V_q by minimizing:

$$\min_{V_q} \sum_{i=0}^{N} \|(x_i - \bar{x}) - V_q V_q^T(x_i - \bar{x})\|_2^2 \tag{1.15}$$

Assuming $\bar{x} = 0$, we can obtain the projection matrix $H_q = V_q V_q^T$ by Singular Value Decomposition (SVD) of the data matrix X:

$$X = UDV^T \tag{1.16}$$

where U is a $N \times p$ orthogonal matrix, its columns are the *left singular vectors*, and V is a $p \times p$ diagonal matrix with ordered diagonal values called the *singular values* The columns of UD are the *principal components*. The solutions to the minimization problem are the first q principal components. The singular values are proportional to the variance they explain, so they can be used to decide how many components to keep obtaining a specific amount of explained variance.

An alternative method is to apply eigen decomposition to the covariance matrix of the data. It yields the same results, but it is sometimes numerically unstable, so the SVD method is preferred.

An intuitive explanation of this method would be to distribute the variance of the data on all existing dimensions by finding the directions that are perpendicular and minimize the projection of the data. To find the first direction we begin with a vector on a random direction that passes through the center of mass of the data, and we rotate the vector until the sum of the projections of the data is maximum, this would be the first component. After that we take a perpendicular vector to all the previous vectors, and we repeat the rotation maximizing the projection until we obtain all the components. Figure 1.9 shows this intuition for a two-dimensional data. In this case the second component is determined by the perpendicularity constraint, so no further computation is needed.

PCA variations

PCA is a linear transformation of the data, this means that if data is linearly separable, the reduced dataset will also be linearly separable (given enough components). We can use the *kernel trick* to map the original attributes to a space where non linearly separable data becomes linearly separable (as in Support Vector Machines). This method is known as *Kernel PCA*. Distances among the examples are defined as a dot product that can be obtained using a kernel:

$$d(x_i, x_j) = \Phi(x_i)^T \Phi(x_j) = K(x_i, x_j) \tag{1.17}$$

Different kernels can be used to perform the transformation to the feature space (polynomial, gaussian. . .).

The computation of the components is equivalent to PCA but performing the eigen decomposition of the covariance matrix computed for the transformed examples:

$$C = \frac{1}{M} \sum_{j=1}^{M} \Phi(x_j)\Phi(x_j)^T \tag{1.18}$$

The main advantage of this method is that it allows discovering patterns that are non-linearly separable in the original space. Also, this method can be applied using any kind of kernel, this means that the features do not have to be continuous or even gaussian. This extends the method to structured data, like for instance graphs or strings.

PCA transforms data to a space of the same dimensionality (all eigenvalues are non-zero) Another variation of PCA is to solve the minimization problem posed by the reconstruction error using ℓ-1 norm regularization, this in known as *sparse PCA*. A penalization term is added to the objective function proportional to the norm of the matrix of eigenvalues:

$$\min_{U,V} \|X - UV\|_2^2 + \alpha \|V\|_1 \tag{1.19}$$

The ℓ-1 norm regularization will encourage sparse solutions (zero eigenvalues).

Figure 1.10 shows PCA and the two described PCA variations applied to the iris dataset. It is interesting to notice in this example, that the representation with the principal components does not differ much from the representation with only two features (Sepal Length, Sepal Width) shown in figure 1.7. In this case just two dimensions are enough for discriminating the groups in the data.

Multidimensional Scaling

The methods in this class of transformations obtain a matrix that projects the dataset directly from M to N dimensions, preserving pairwise data distances. This results in a new dataset that maintains only the distance relations between pairs of examples, but usually this is enough to uncover the structure in the data.

The method for computing the projection matrix is based on the optimization of a function of the pairwise distances (called stress function). This means that the actual attributes are not

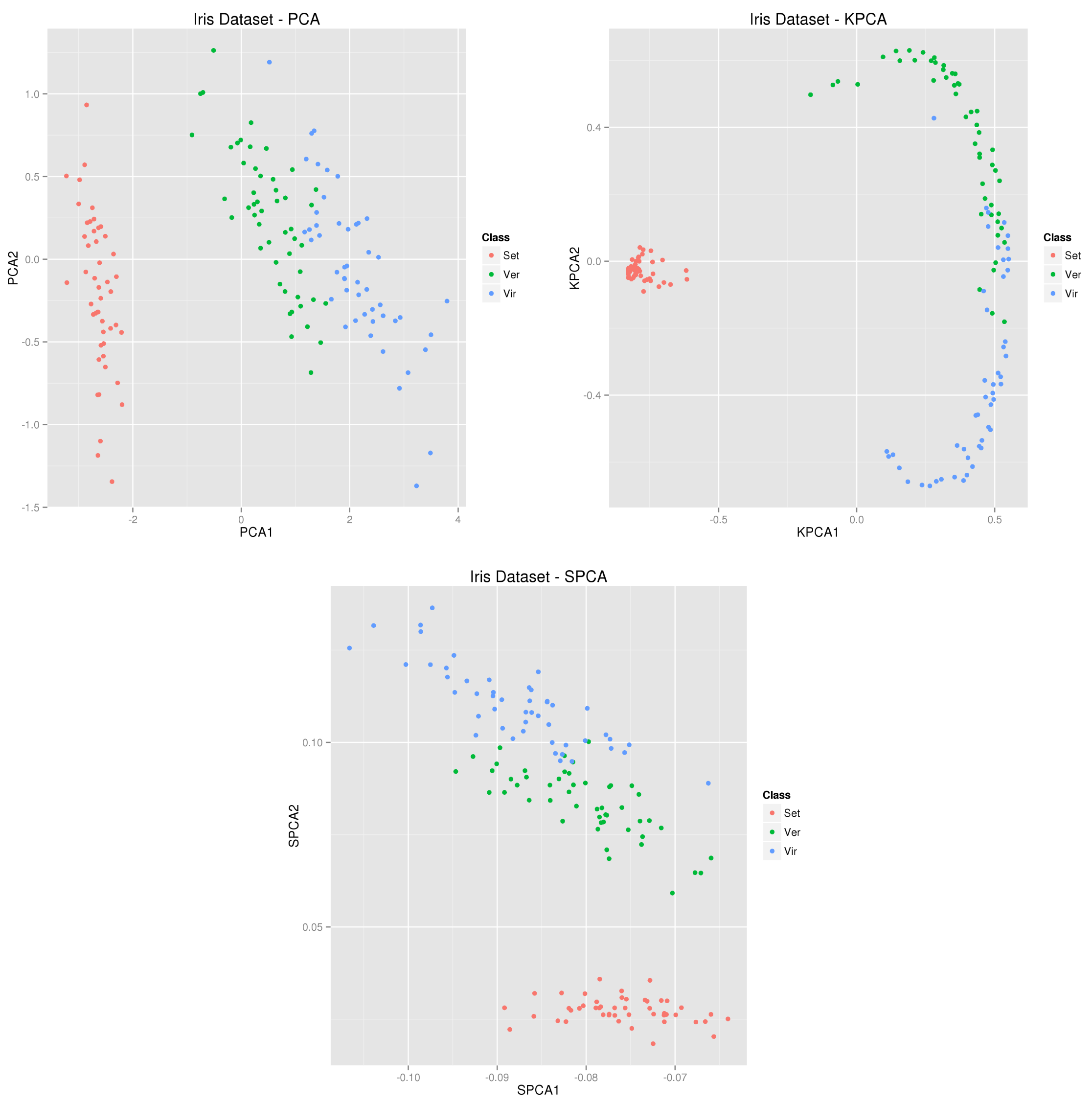


Figure 1.10: PCA, Kernel PCA (RBF kernel) and Sparse PCA transformation of the iris dataset

used in the transformation, only the distances, this makes the method suitable for any data where a distance function can be defined, independently of the representation of the attributes (they do not have to be continuous features).

This method allows the optimization of different objective functions (distortion) that leads to several variants, such as *least squares MDS*, *Sammong mapping* or *classical scaling*. The optimization problem is usually solved by gradient descent.

For example, for Least Squares Multidimensional Scaling (MDS) the distortion is defined as the sum of the square distance between the original distance matrix and the distance matrix of the new data:

$$S_D(z_1, z_2, \ldots, z_n) = \left[\sum_{i \neq i'} (d_{ii'} - \|z_i - z_{i'}\|_2)^2 \right] \tag{1.20}$$

The optimization problem is defined as:

$$\arg\min_{z_1, z_2, \ldots, z_n} S_D(z_1, z_2, \ldots, z_n) \tag{1.21}$$

Several strategies can be used to solve this problem. If the distance matrix is euclidean, it can be solved applying eigen decomposition just like PCA. In other cases gradient descent can be used with the derivative of $S_D(z_1, z_2, \ldots, z_n)$ and a step α in the following fashion:

1. Begin with a guess for Z
2. Repeat until convergence:

   $$Z^{(k+1)} = Z^{(k)} - \alpha \nabla S_D(Z)$$

The other variations of MDS change the characteristics of the new space to highlight the patterns. For instance, Sammong Mapping puts emphasis on smaller distances, using as objective function:

$$S_D(z_1, z_2, \ldots, z_n) = \left[\sum_{i \neq i'} \frac{(d_{ii'} - \|z_i - z_{i'}\|)^2}{d_{ii'}} \right] \tag{1.22}$$

Differently, Classical Scaling uses a similarity function instead of a distance, with the following objective function:

$$S_D(z_1, z_2, \ldots, z_n) = \left[\sum_{i \neq i'} (s_{ii'} - \langle z_i - \bar{z}, z_{i'} - \bar{z} \rangle)^2 \right] \tag{1.23}$$

The final value of the optimized function is a measure of how close are the relative distances in the new space to the original one and can be used as a criterion to decide how many dimensions to use for the transformation.

All these three methods assume an euclidean space, and are called metric scaling methods. The actual distances are approximated on the lower rank space.

There are datasets where there is only a ranking among the examples, and the distances are qualitative indicating only an order, and the magnitude of the distance is unknown or irrelevant. For these datasets non-metric MDS is applied.

In this case the objective function is:

$$S_D(z_1, z_2, \ldots, z_n) = \frac{\sum_{i,i'} [\theta(||z_i - z_{i'}||) - d_{ii'}]^2}{\sum_{i,i'} d_{i,i'}^2} \tag{1.24}$$

where $\theta(\cdot)$ is an arbitrary increasing function, $d_{i,i'}$ is a ranking function that orders the distance among the pairs of examples.

Non-negative Matrix Factorization

Non-negative Matrix Factorization (NMF) uses a similar idea than PCA, the decomposition of the data matrix into the product of two matrices. The main difference is that the values of the matrices are constrained to be positive. This formulation assumes that the data is the sum of unknown positive latent variables, and the positiveness assumption helps to the interpretation of the result.

The optimization problem for NMF is defined as, given a number of target dimensions $k < \min((m, n)$ (m the number of examples, n the number of features), to find the pair of non-negative matrices $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{k \times n}$ that minimize

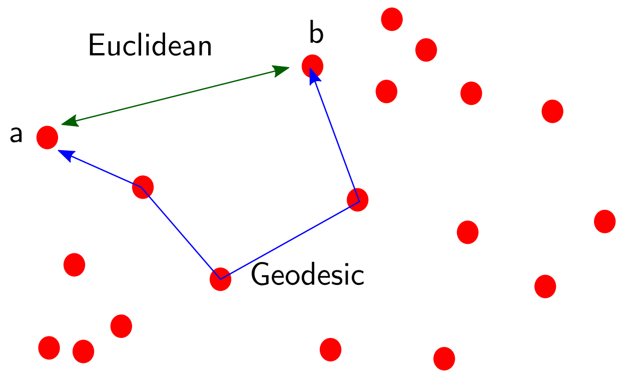


Figure 1.11: Euclidean distance versus geodesic distance

$$f(W,H) = \frac{1}{2}||A - WH||_F^2 \tag{1.25}$$

This problem is not convex, and it must be solved using an iterative optimization algorithm. The most common algorithm used is based on *alternate least squares*. The idea is to initialize the algorithm with two random matrices and iteratively perform two steps, fixing one of the matrices and finding the least squares solution for the other. Each step the negative values that could appear on the matrix that is being computed are zeroed.

Some variations of NMF exists like adding a sparsity constraint using the ℓ-1 norm over the sum of the elements of the matrices or forcing the orthogonality of the matrices (like the solution from PCA)

This is interesting in domains like, for instance, text mining, where it can be assumed that documents can be described as the composition of topics, or image recognition, where an image can be obtained by joining object features, or recommender systems, where a recommendation depends on topics of interest of the users.

This area is a hot topic because of its practical applications and many methods exists, a recent review can be found in [135].

ISOMAP

This method computes a non-linear transformation between the original space and the reduced new space. It is assumed that there is a low dimensional embedding of the data able to represent the relations among the patterns in the data. The geodesic distance among the examples is used for this transformation instead of the usual euclidean distance. This puts more emphasis in local densities. This distance is computed as the length of the shortest path that connects two nodes in the neighborhood graph of the data. Figure 1.11 shows an example comparing both distances. Examples a and b are in different dense areas but they are relatively close according to euclidean distance, geodesic distance is more aware of the different densities.

The choice of this measure is based on the assumption that the relation of an instance with its immediate neighbors holds the most representative information about the structure of the data. Obtaining a new space that preserves this distance would preserve these neighborhood relations. In fact, we are obtaining a new space where the locality of the data behavior is preserved.

The algorithm for computing this transformation is as follows:

1. For each data point find its k-closest neighbors (points at minimal euclidean distance)
2. Build a graph where each point has an edge to its closest neighbors

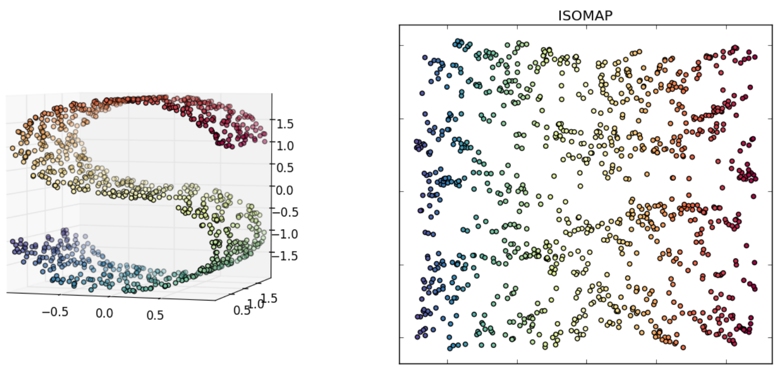


Figure 1.12: ISOMAP

3. Approximate the geodesic distance for each pair of points by the shortest path in the graph (using Kruskal algorithm for example)

4. Apply a MDS algorithm to the distance matrix of the graph to obtain the new set of coordinates

Locally Linear Embedding

Locally Linear Embedding [38, 113, 144] performs a transformation that preserves the local structure of the data. The idea is to maintain the distances among examples assuming that the original space is locally linear and to relax the distances among distant examples.

It assumes that each example can be reconstructed by a linear combination of its k neighbors. This represents the linear locality of the space. We have to tune this hyperparameter having in mind that the larger its value, the larger the locality, so it can degenerate to a linear transformation. This value also constraints the number of dimensions that the target transformation can have, that must be less than the number of the original dimensions minus one ($k < D - 1$).

The method computes for each example a set of reconstruction weights from its k-nearest neighbors using least squares. From these weights a new set of coordinates in a lower dimensional space is computed for each data point that is consistent with the weights computed in the original data using eigen decomposition. Different variants of this algorithm exist that address some problems that the original algorithm presents.

The original algorithm for this method is as follows:

1. For each data point find the K-nearest neighbors in the original p-dimensional space ($\mathcal{N}(i)$)

2. Approximate each point by a mixture of the neighbors, solving the optimization problem:

$$\min_{W_{ik}} \|x_i - \sum_{k \in \mathcal{N}(i)} w_{ik} x_k\|^2 \tag{1.26}$$

   where $w_{ik} = 0$ if $k \notin \mathcal{N}(i)$, with $\sum_{k \in \mathcal{N}(i)} w_{ik} = 1$ and $K < p$

3. Find points y_i in a space of dimension $d < p$ that minimize:

$$\sum_{i=0}^{N} \|y_i - \sum_{k \in \mathcal{N}(i)} w_{ik} y_k\|^2 \tag{1.27}$$

In order to decide the number of dimensions adequate for the transformation, the reconstruction error can be used to compare the different alternatives.

Local MDS

This is a non-linear version of MDS that uses an objective function that gives more emphasis to local distances. This locality is also derived from the neighborhood graph that is used in the computation of the objective function. The idea is to obtain a new space where neighbor examples are closer, and the distances to non neighbors are magnified, improving separability.

Given a set of pairs of points $\mathcal{N}$ where a pair (i,i') belongs to this set if i is among the k neighbors of i' or vice versa, the objective function that is minimized is:

$$S_L(z_1, z_2, \ldots, z_N) = \sum_{(i,i')\in\mathcal{N}} (d_{ii'} - \|z_i - z_{i'}\|)^2 - \tau \sum_{(i,i')\notin\mathcal{N}} (\|z_i - z_{i'}\|) \tag{1.28}$$

The parameter τ controls how much the non neighbors are scattered.

Random Projections

All the previous methods are computationally expensive because a projection/transformation matrix has to be obtained to perform the dimensionality reduction with the corresponding computational cost. Random projections [13, 85] take advantage of the properties of certain transformation matrices to obtain a reasonable approximation with a reduced computational cost.

The idea is that if we generate randomly a transformation matrix with certain conditions, it is obtained a projection to a new space of reduced dimensionality that maintains the distances among the examples in the original space.

The matrix generated has as dimensions $n\_features \times n\_components$, being $n\_components$ the number of dimensions of the new space

There are two main strategies for generating the random matrix:

- To generate the matrix using a gaussian distribution $N(0, \frac{1}{n\_components})$, this matrix is dense.
- To generate a sparse matrix of certain density (s) with values:

$$\begin{cases} \sqrt{\frac{s}{n\_components}} & & \frac{1}{2s} \\ 0 & with\ \ probability & 1-\frac{1}{s} \\ -\sqrt{\frac{s}{n\_components}} & & \frac{1}{2s} \end{cases}$$

These matrices are almost orthogonal (close to a transformation matrix obtained by PCA). The effectiveness of the transformation usually depends on the number of dimensions, the Johnson-Lindenstrauss lemma ([26]) can be used to obtain an estimate of the number of dimensions from the number of examples in the dataset, and the expected distortion. This lemma states that given $0 < \epsilon < 1$ and a set $\mathcal{X}$ of m points in $\mathbb{R}^N$ a linear mapping $f : \mathbb{R}^N \to \mathbb{R}^n$ of the points that holds:

$$(1-\epsilon)||u-v||^2 \leq ||f(u) - f(v)||^2 \leq (1+\epsilon)||u-v||^2 \tag{1.29}$$

needs a number of dimensions that is bounded by the following inequality:

$$n > \frac{8\log(m)}{\epsilon^2} \tag{1.30}$$

t-SNE

Another dimensionality reduction technique that has been gaining relevance lately is t-Stochastic Neighbor Embedding (t-SNE, [89]). Actually it is presented as a visualization technique more than a dimensionality reduction method. It is claimed to have much better results than other methods like ISOMAP, LLE or MDS.

This method assumes that distances are expressed as probabilities using a gaussian kernel as density estimation. This method optimizes the Kullback-Leiber divergence between the original data distances distribution and the distribution of the distance of the transformed data. The main idea is to obtain a lower dimensionality manifold that reproduces the data distribution from the original data space.

The basic steps of the algorithm are:

1. For each example of the dataset, the distance to the rest of the data is computed and normalized so all the distances for an example sum one
2. A similarity matrix is computed representing the distribution of the examples
3. Data is initially projected to a lower dimensionality space, and their distances are computed and normalized as before
4. A similarity matrix is computed and the KL divergence is computed
5. Data points are moved in order to improve the KL-divergence
6. The last two steps are repeated until no further improvement is obtained

Unfortunately, the optimization problem defined is non-convex, there is also a dependence on the bandwidth of the gaussian kernel used for the density estimation, and the initial number of dimensions can affect the result. All that means that different initialization and parameters can result in very different solutions. This link[1] ([136]) gives some insights about how to tune its parameters. An option is to initialize the algorithm with a subset of the features obtained computing PCA to obtain a similar solution.

From its original version, several improvements have been introduced to obtain high scalability using different approximations and now is used broadly as a visualization tool.

Autoencoders

Autoencoders are fully connected neural networks that are trained to reproduce their input in their output. This process allows learning a representation of the data that can be used for visualization, or as a representation for other learning tasks.

This kind of neural network is composed by two parts, an *encoder* that transforms the data into a possibly lower dimensional space, generating a set of latent features and a *decoder* that reconstructs the original data from the latent representation. The representation will be the result of the encoder.

The goal is to optimize a loss function that measures how close are the input and the output, considering $f(x)$ the function that is learned by the encoder and $g(x)$ the function learned by the decoder, the loss function is:

$$\mathcal{L}(x) = L(x, g(f(x))) \tag{1.31}$$

with for instance L as the mean squared difference between the original and transformed data.

[1] In case you are reading this notes on paper the link is `https://distill.pub/2016/misread-tsne/`

The network needs to have specific characteristics so a trivial solution or an overspecialized representation is not learned. This can be accomplished in different ways. For instance, a network with an encoder that progressively reduces the size of the layers and a decoder that increases progressively the size of the layers will be able to learn a lower dimensional representation with as many dimensions as the last layer of the decoder. With this architecture and using linear activation functions the space that is learned corresponds to the same space as PCA, but in this case the number of dimensions that we obtain is determined by the size of the last layer of the decoder.

When non-linear activation functions are used, more complex representations can be learned, but it will be easy to overfit, with the possibility of just memorizing the data depending on the capacity of the network. This arises the need for other techniques for avoiding this problem, like, for instance, regularization techniques that constraint the parameters of the network in some way. This also allows having architectures that learn representations with the same or more dimensions that the input data obtaining more complex representations with some latent semantic information.

An alternative to regularization is to change the elements used on the error function. For instance, *denoising autoencoders* learn to reconstruct the data from corrupted examples instead of from the original data, so the values used on the error function that are transformed using the encoder and the decoder are the noisy examples. The autoencoder learns how to reproduce the denoised input from the latent representation and does not rely on simple statistical features or too specific information from the training data.

1.3.3 Notebook: Dimensionality Reduction

This notebook shows some of the dimensionality reduction algorithms explained in the course.

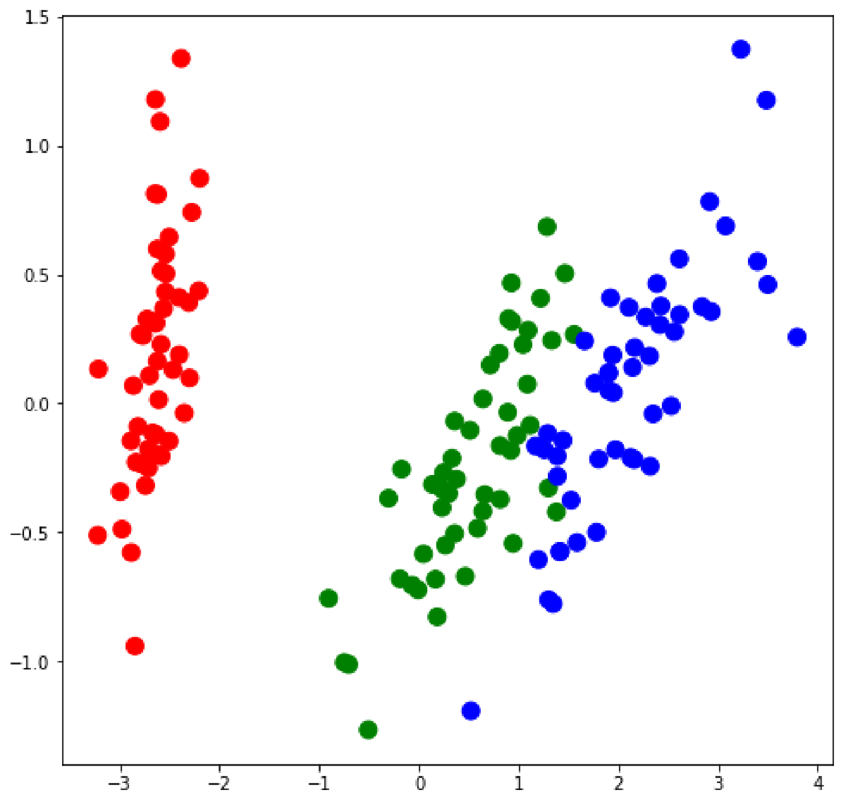
[38]:
```
from sklearn import datasets
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
from sklearn.decomposition import KernelPCA, PCA
from sklearn.manifold import LocallyLinearEmbedding, Isomap, TSNE
from kemlglearn.feature_selection.unsupervised import LaplacianScore

from kemlglearn.datasets import make_blobs
from sklearn.datasets import make_moons
import keras
from keras import layers

iris = datasets.load_iris()
col = ['r', 'g', 'b']
lc = [col[i] for i in iris['target']]
```

Dimensionality Reduction - Linear: PCA

We bill begin with PCA and the iris dataset. Given that the clusters in this dataset are linearly separable, this kind of transformation will allow to represent the clusters in the data given enough components.

[39]:
```
pca = PCA()
pdata = pca.fit_transform(iris['data'])
```

```
i=0
j=1

fig = plt.figure(figsize=(8,8))
plt.scatter(pdata[:, i], pdata[:, j], c=lc,s=100);
```



Looking at the variance explained by each component, we can see that the first one has most of the variance and the two first components explains almost 98%, so only this components would be necessary and the 2D visualization of the data will represents very closely the original data.

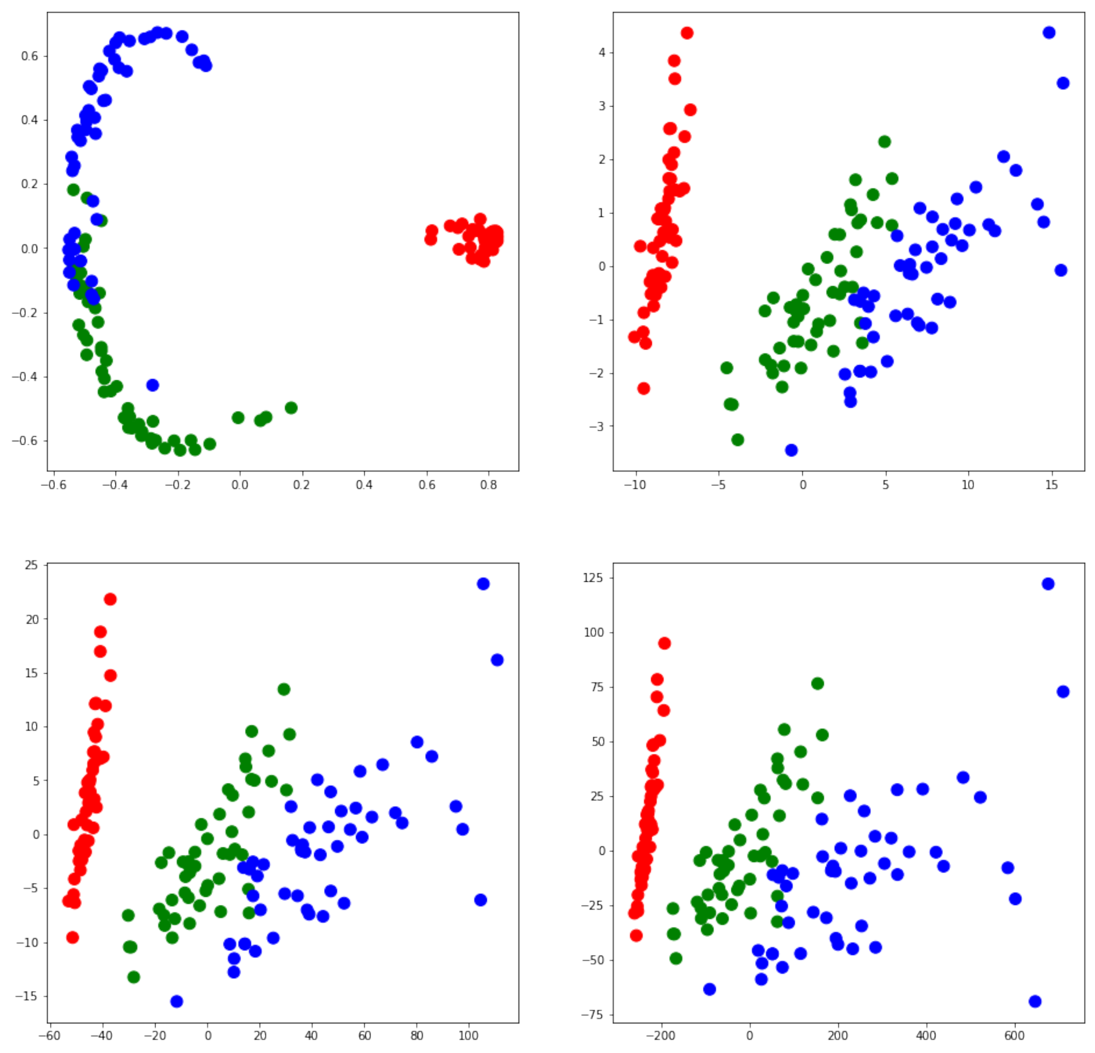
[40]:
```
print(pca.explained_variance_ratio_)
```

```
[0.92461872 0.05306648 0.01710261 0.00521218]
```

Dimensionality Reduction - non Linear: Kernel PCA

Even when a linear transformation is enough for the iris data we can check what happens when non linear transformations are applied, we will start with Kernel PCA applying RBF kernel and different degrees polynomial kernels

[41]:
```
kernel=['rbf', 'poly', 'poly', 'poly']
degree=[1, 2,3,4]

fig = plt.figure(figsize=(16,16))
for i, (k, d) in enumerate(zip(kernel, degree)):
    ax = fig.add_subplot(2,2,i+1)
    kpca = KernelPCA(n_components=2, kernel=k, degree=d)
    kpdata = kpca.fit_transform(iris['data'])
    plt.scatter(kpdata[:, 0], kpdata[:, 1], c=lc,s=100);
```

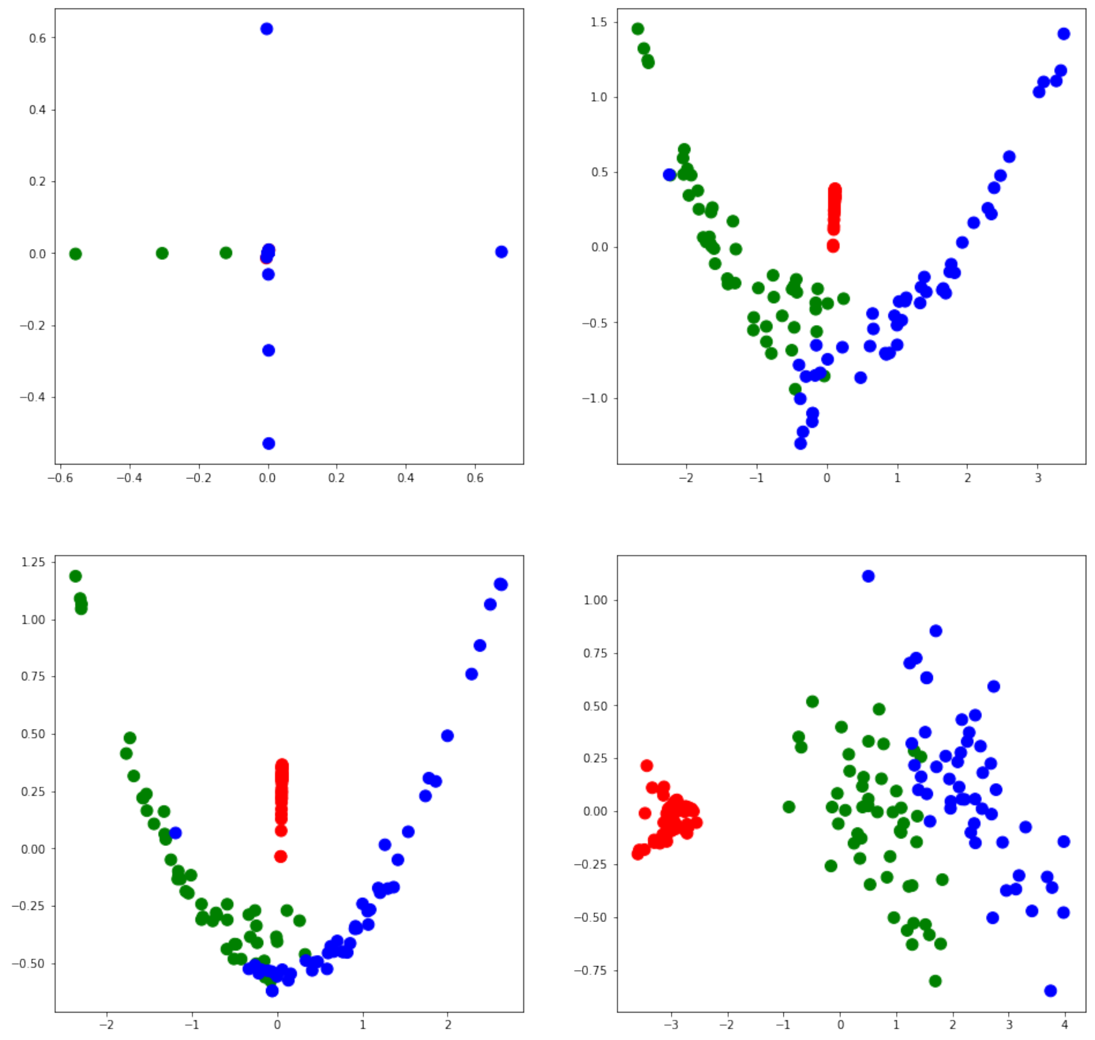


As we can see polynomial kernels do not results in a much different representation of the data apart from putting closer the groups as the degree increases. The RBF kernel does a poor job maintaining the separability of the green and blue classes, sometimes to apply a nonlinear transformation can change some of the relations among the examples

Dimensionality Reduction - non Linear: ISOMAP

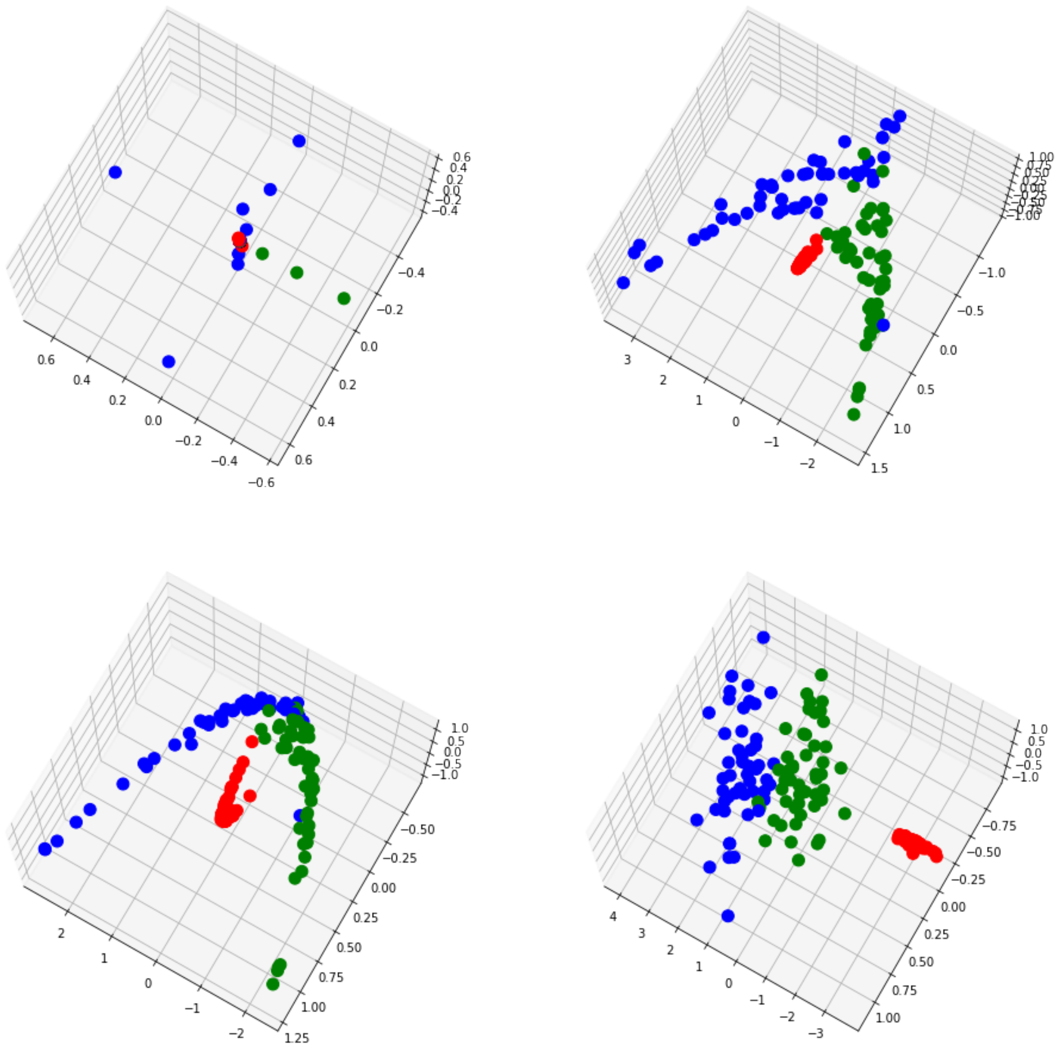
ISOMAP is also a nonlinear transformation but the final representation will depend on the number of neighbors used to compute the distance graph as we can see

[42]:
```
nneigh=[1, 5, 21, 41]

fig = plt.figure(figsize=(16,16))
for i, nn in enumerate(nneigh):
    ax = fig.add_subplot(2,2,i+1)
    iso = Isomap(n_components=2, n_neighbors=nn)
    isdata = iso.fit_transform(iris['data'])
    plt.scatter(isdata[:, 0], isdata[:, 1], c=lc,s=100);
```



Increasing the number of components can give more freedon to the algorithm to allocate the local densities

```
[43]: nneigh=[1, 5, 21, 41]

      fig = plt.figure(figsize=(16,16))
      for i, nn in enumerate(nneigh):
          iso = Isomap(n_components=3, n_neighbors=nn)
          is3data = iso.fit_transform(iris['data'])
          ax = fig.add_subplot(2,2,i+1, projection='3d')
          ax.view_init(75, 120)
          plt.scatter(is3data[:, 0], is3data[:, 1], zs=is3data[:, 2],␣
       ↪depthshade=False, c=lc,s=100);
```

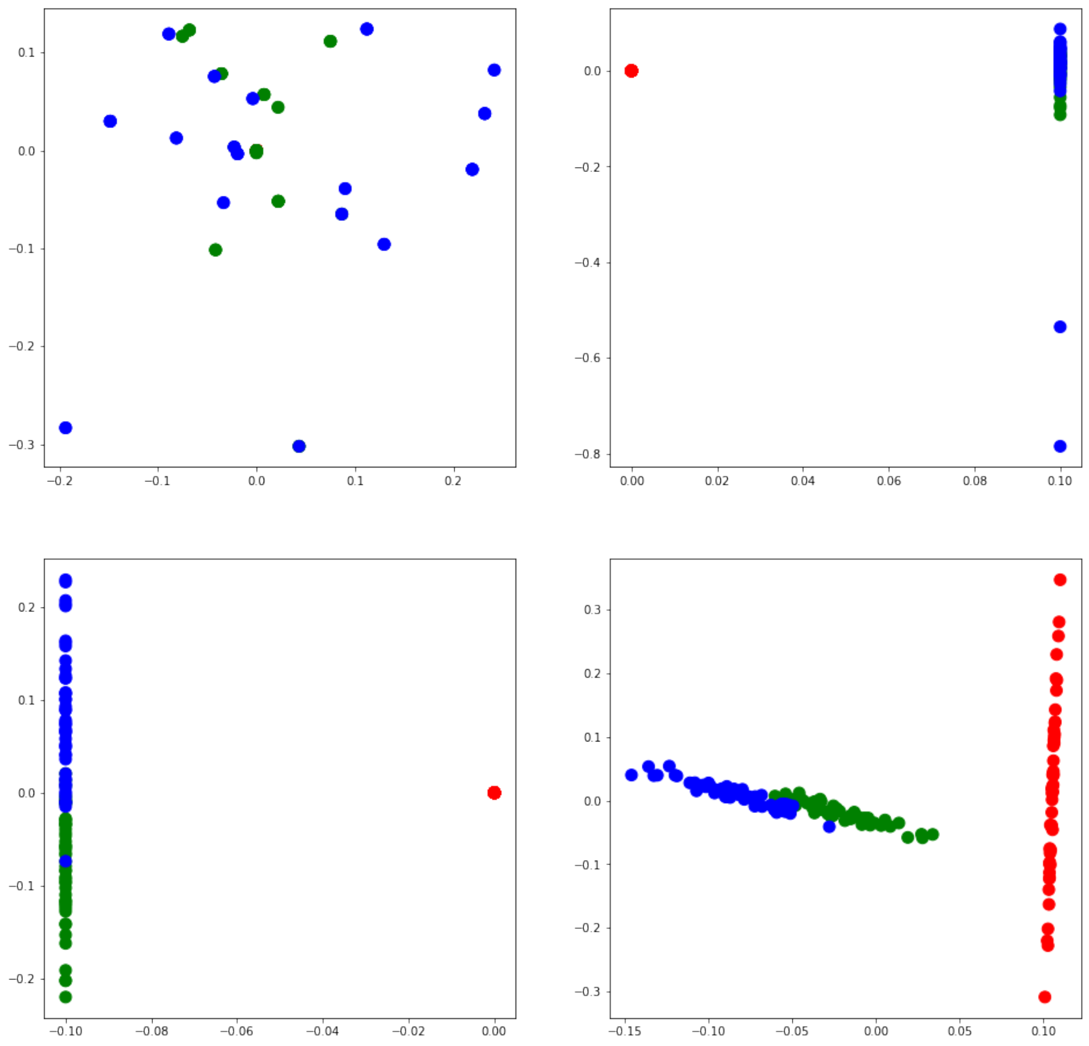


Dimensionality Reduction - non Linear: LLE

LLE uses the local reconstruction and results in a very different representation than ISOMAP, also the number of neighbors have an impact in the results and we can see that the reconstruction error increases with the number of neighbors.

[44]:
```python
nneigh=[1, 5, 21, 41]

fig = plt.figure(figsize=(16,16))
for i, nn in enumerate(nneigh):
    lle = LocallyLinearEmbedding(n_neighbors=nn, n_components=2, method='standard')
    lldata = lle.fit_transform(iris['data'])
    ax = fig.add_subplot(2,2,i+1)
    print (f'Rec Error {nn} = {lle.reconstruction_error_}')
    plt.scatter(lldata[:, 0], lldata[:, 1], c=lc,s=100);
```

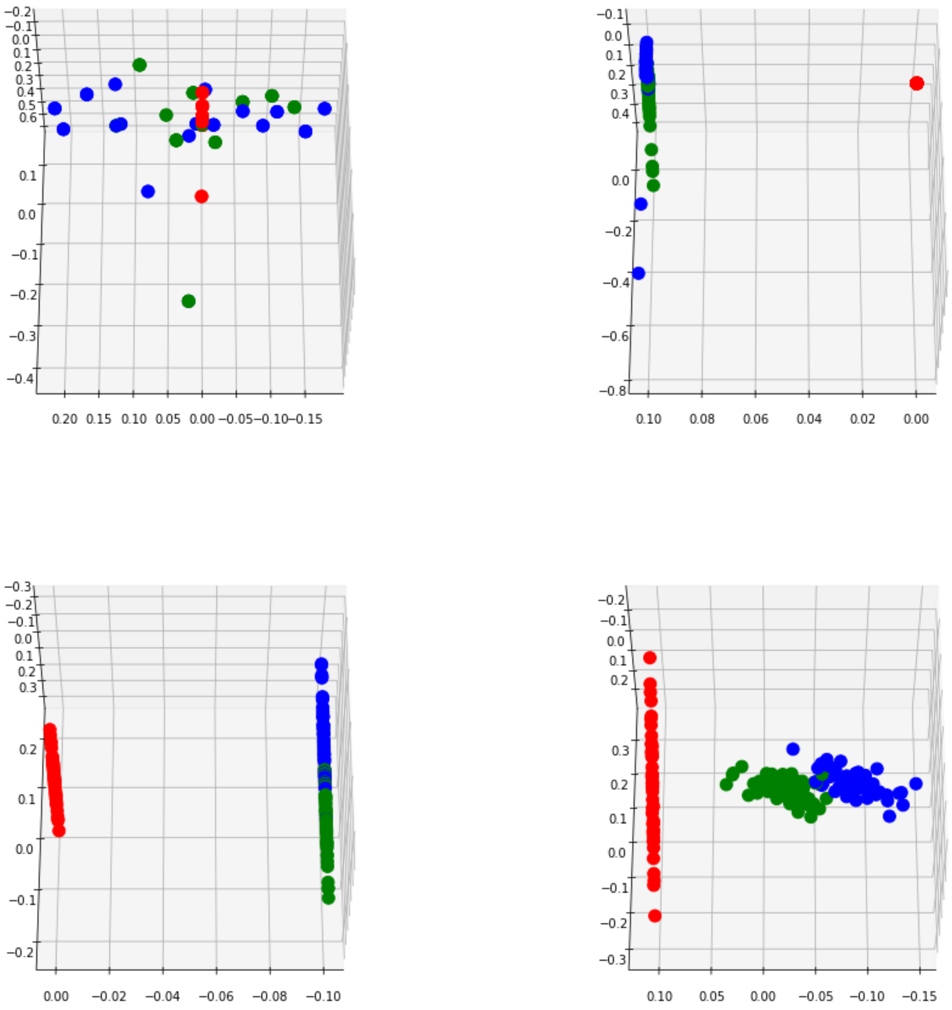
```
Rec Error 1 = -3.004051247345241e-15
Rec Error 5 = 6.641420753574737e-08
Rec Error 21 = 4.321606693213067e-06
Rec Error 41 = 1.8567053734961027e-05
```



Also the number of dimensions can help to allocate better the local densities

```
[45]: nneigh=[1, 5, 21, 41]

fig = plt.figure(figsize=(16,16))
for i, nn in enumerate(nneigh):

    lle = LocallyLinearEmbedding(n_neighbors=nn, n_components=3,␣
 ↪method='standard')
    ll3data = lle.fit_transform(iris['data'])
    print (f'Rec Error {nn} = {lle.reconstruction_error_}')
    ax = fig.add_subplot(2,2,i+1, projection='3d')
    ax.view_init(240, 90)
    plt.scatter(ll3data[:, 0], ll3data[:, 1], zs=ll3data[:, 2],␣
 ↪depthshade=False, c=lc,s=100);
```

```
Rec Error 1 = -5.010373939594841e-15
Rec Error 5 = 3.992905786848256e-07
Rec Error 21 = 1.955534635062646e-05
Rec Error 41 = 5.4587047544793695e-05
```

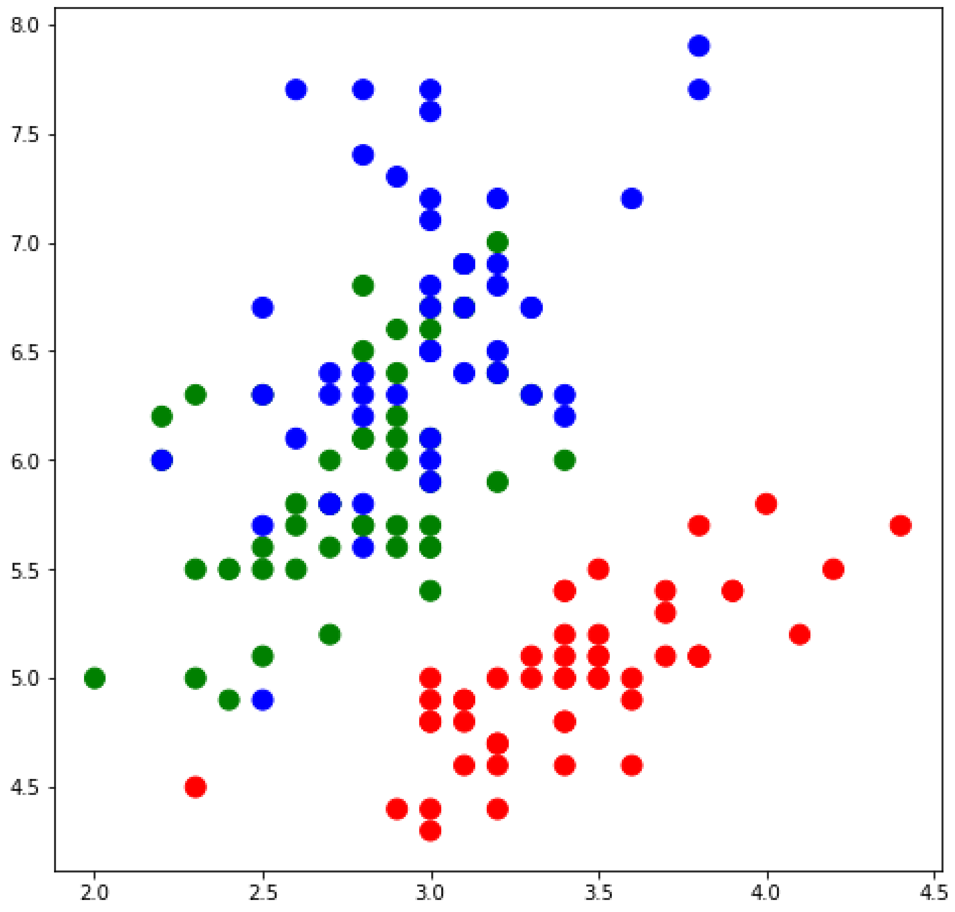


Feature Selection: Laplacian Score

Unsupervised feature selection is a more difficul task, Laplacian Score measures how the different original dimensions contribute to the local densities and returns a score proportional to that contribution.

[46]:
```python
nneigh=[1, 5, 21, 41]

for i, nn in enumerate(nneigh):
    lap = LaplacianScore(n_neighbors=nn, bandwidth=0.1, k=2)
    irissel = lap.fit_transform(iris['data'])
    print (f'Laplacian Score {nn}= {lap.scores_}')
fig = plt.figure(figsize=(8,8))
plt.scatter(irissel[:, 0], irissel[:, 1], c=lc,s=100);
```

```
Laplacian Score 1= [0.06905246 0.24742775 0.03309567 0.0538684 ]
Laplacian Score 5= [0.13637993 0.39333255 0.05770985 0.05221089]
Laplacian Score 21= [0.36486037 0.59757257 0.14479276 0.06950606]
Laplacian Score 41= [0.49181376 0.7508564  0.21223769 0.18608088]
```



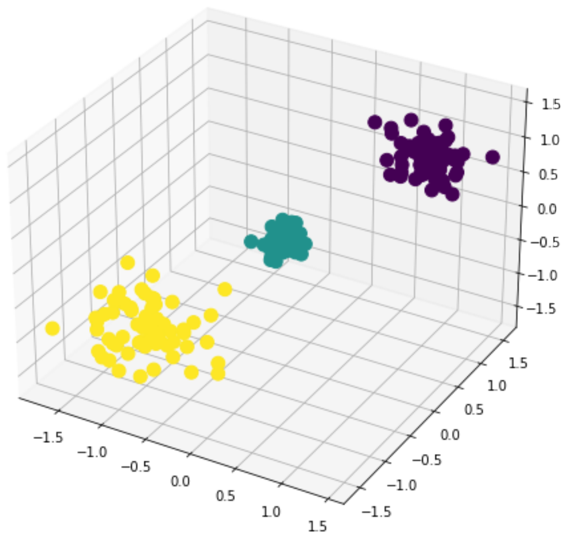
As we can see always there is a variable that has the larger contribution, the second one, followed by the first one. In this case we can see that these two variables do not do the best job separating the clusters, it would be a better idea to apply other method that takes in account the contribution of all the variabes in a new space of reduced dimensionality.

Linear data

Now we are going to use some artificial datasets to see the effect of linear and non linear dimensionality reduction algorithms.

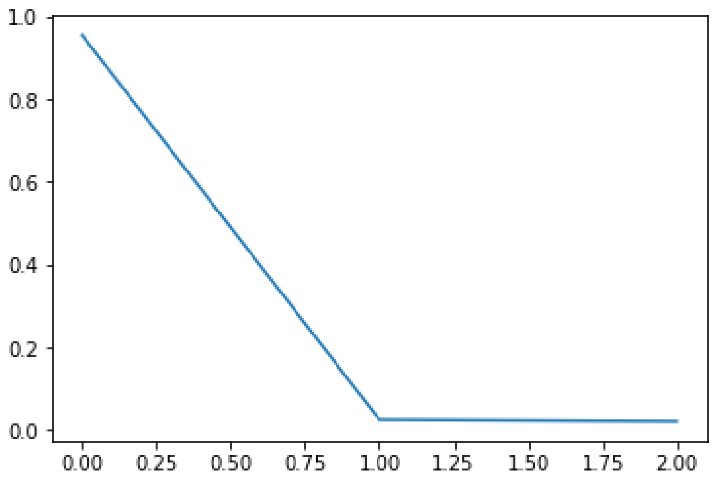
The first one consists of three spherical blobs with different variances.

[47]:
```
blobs, labels = make_blobs(n_samples=200, n_features=3,␣
↪centers=[[1,1,1],[0,0,0],[-1,-1,-1]], cluster_std=[0.2,0.1,0.3])

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
plt.scatter(blobs[:, 0], blobs[:, 1], zs=blobs[:, 2], depthshade=False,␣
↪c=labels, s=100);
```



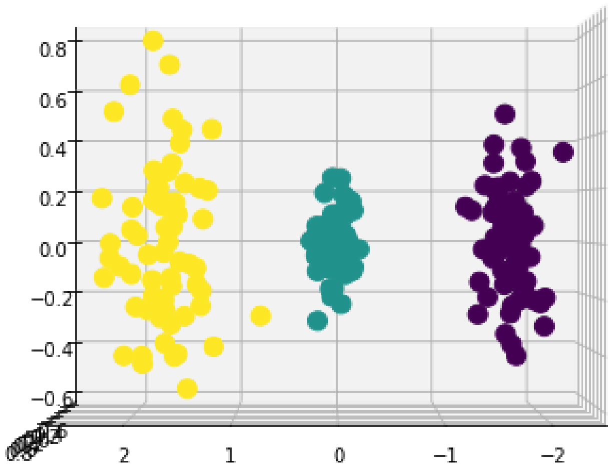
Computing PCA we can see that just one component explains almost all the variance.

```
[48]: pca = PCA()
fdata = pca.fit_transform(blobs)
fig = plt.figure()
plt.plot(range(len(pca.explained_variance_ratio_)), pca.
 ↪explained_variance_ratio_);
```

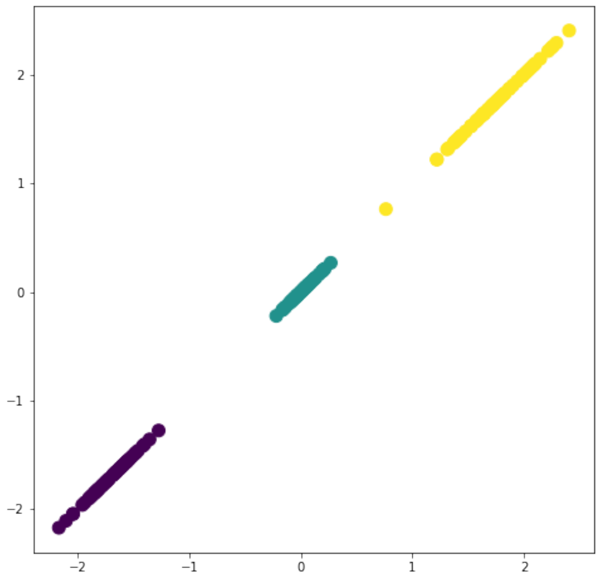


And as we can expect representing the PCA transformed data we can see that just one dimension is enough

```
[49]: fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.view_init(0, 90)
plt.scatter(fdata[:, 0], fdata[:, 1], zs=fdata[:, 2], depthshade=False,␣
 ↪c=labels,s=100);
```



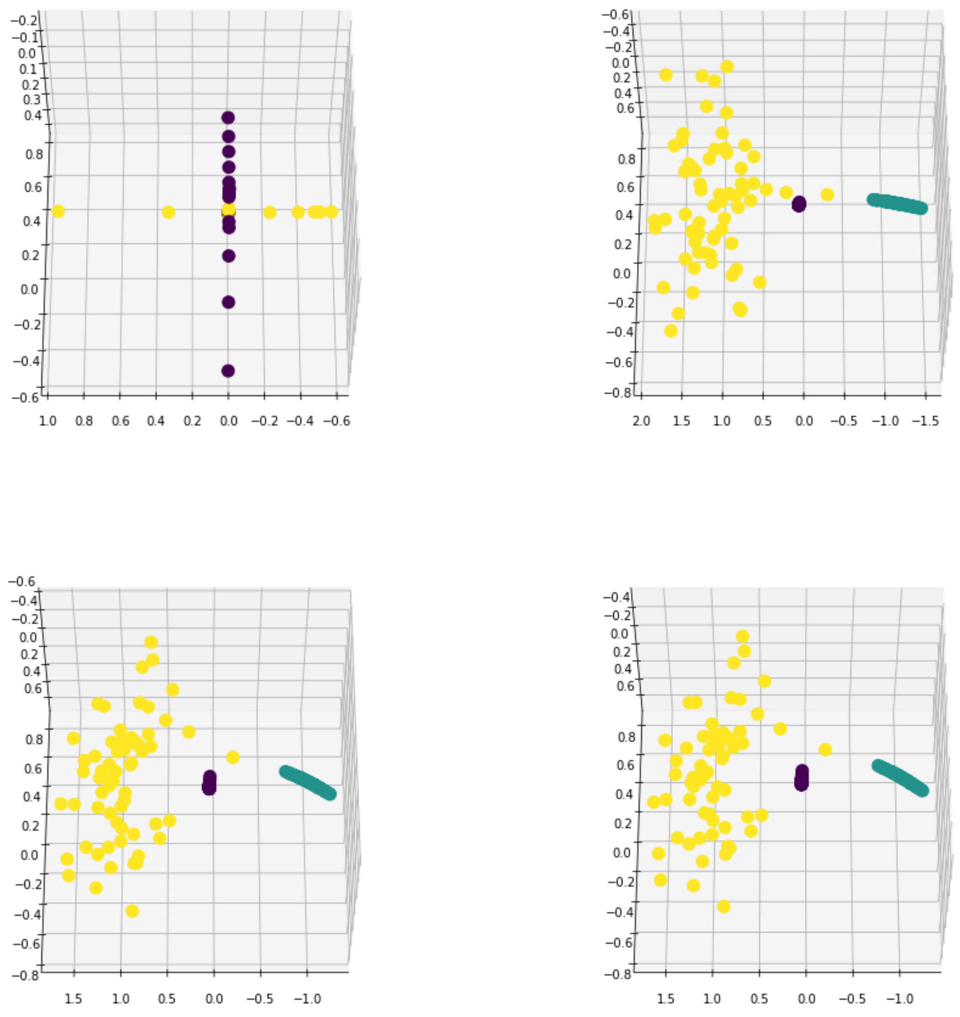
[50]:
```
fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111)
plt.scatter(fdata[:, 0], fdata[:, 0], c=labels,s=100);
```



Using nonlinear transformations to linearly separable data sometimes has weird consequences.

First we see what happens using ISOMAP

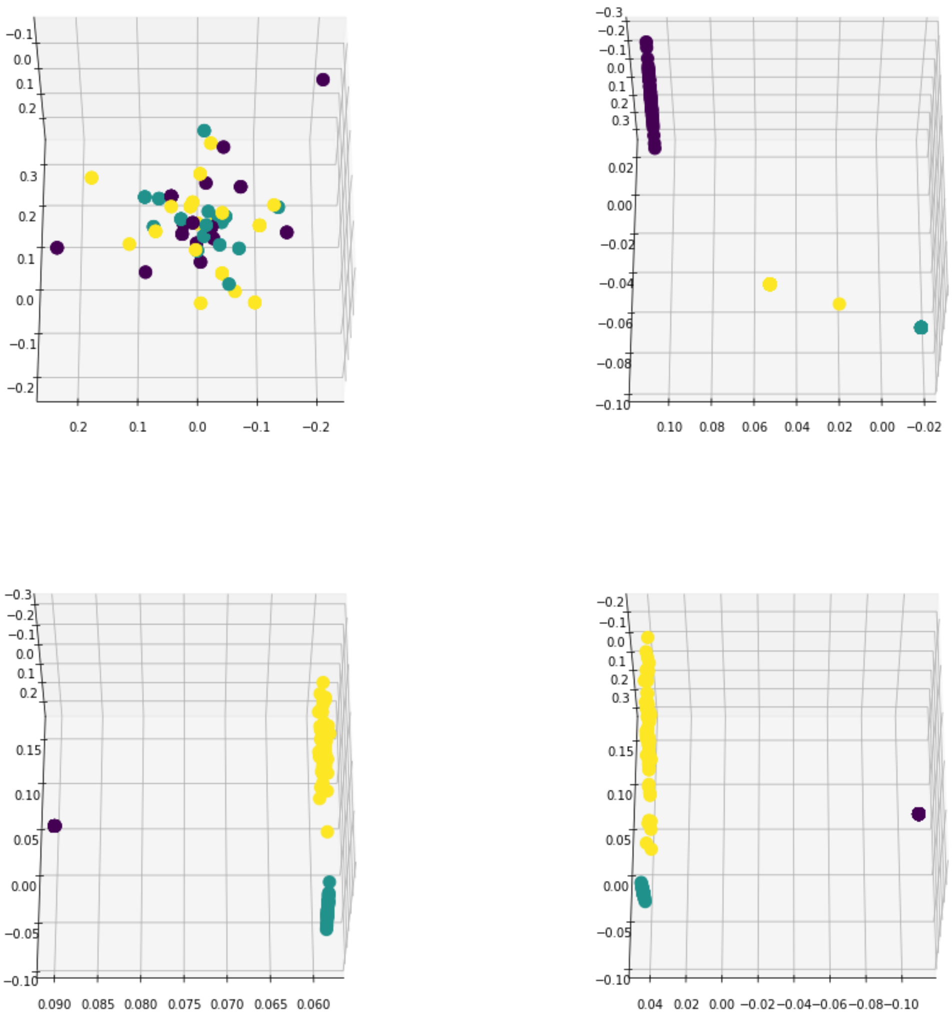
[51]:
```
nneigh=[1, 5, 17, 21]

fig = plt.figure(figsize=(16,16))
for i, nn in enumerate(nneigh):
    iso = Isomap(n_components=3, n_neighbors=nn)
    fdata = iso.fit_transform(blobs)
    ax = fig.add_subplot(2,2,i+1, projection='3d')
    ax.view_init(240, 90)
    plt.scatter(fdata[:, 0], fdata[:, 1], zs= fdata[:, 2], depthshade=False,␣
 ↪c=labels, s=100);
```



Changing the number of neigbors can change radically the result, but as they increase the transformation is equivalent to PCA

Now for Locally Linear Embedding

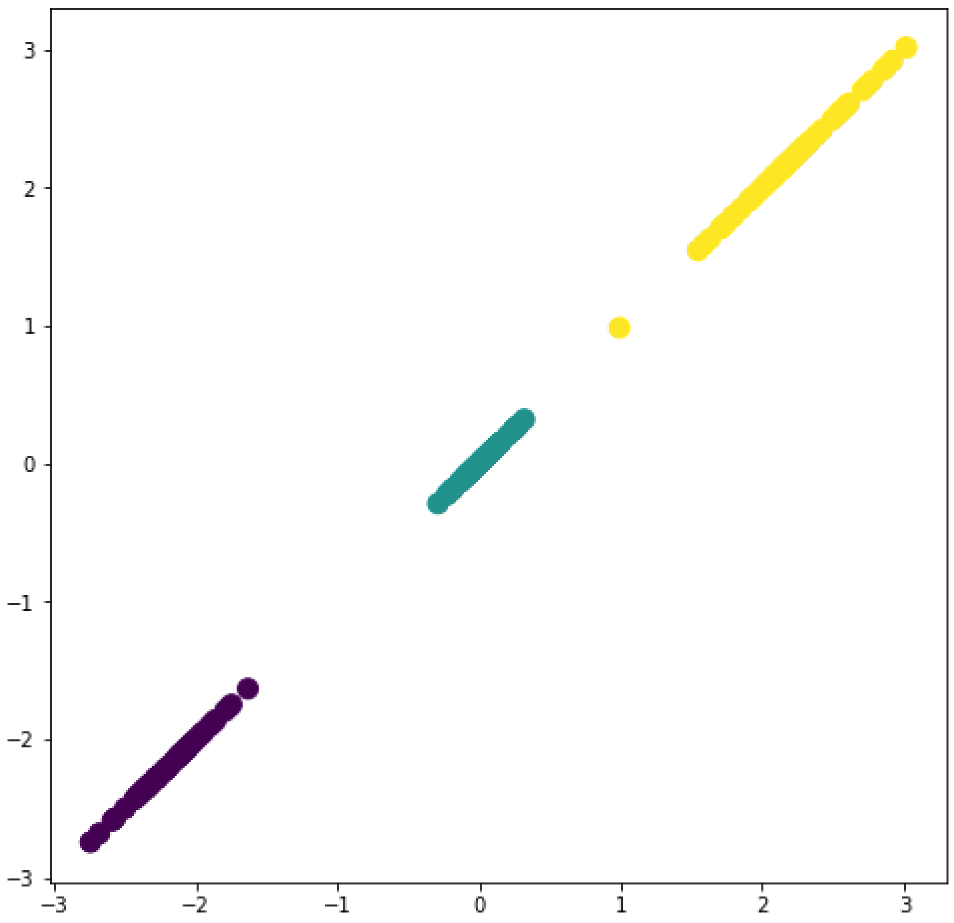
[52]:
```
nneigh=[1, 5, 17, 21]

fig = plt.figure(figsize=(16,16))
for i, nn in enumerate(nneigh):
    lle = LocallyLinearEmbedding(n_neighbors=nn, n_components=3,␣
↪method='standard')
    fdata = lle.fit_transform(blobs)
    ax = fig.add_subplot(2,2,i+1, projection='3d')
    ax.view_init(240, 90)
    plt.scatter(fdata[:, 0], fdata[:, 1], zs= fdata[:, 2], depthshade=False,␣
↪c=labels, s=100);
```



If we train an autoencoder with linear activations and layer sizes 3-2-3, we can obtain a representation that will be equivalent as PCA. After training we only need to keep the encoder for performing the transformation

[53]:
```
input = keras.Input(shape=(3,))
encoded = layers.Dense(2, activation='linear')(input)
decoded = layers.Dense(3, activation='linear')(encoded)

autoencoder = keras.Model(input, decoded)
encoder = keras.Model(input, encoded)
autoencoder.compile(optimizer='adam', loss='mse')
autoencoder.fit(blobs, blobs, epochs=50, batch_size=32, shuffle=True,␣
↪verbose=False);
```

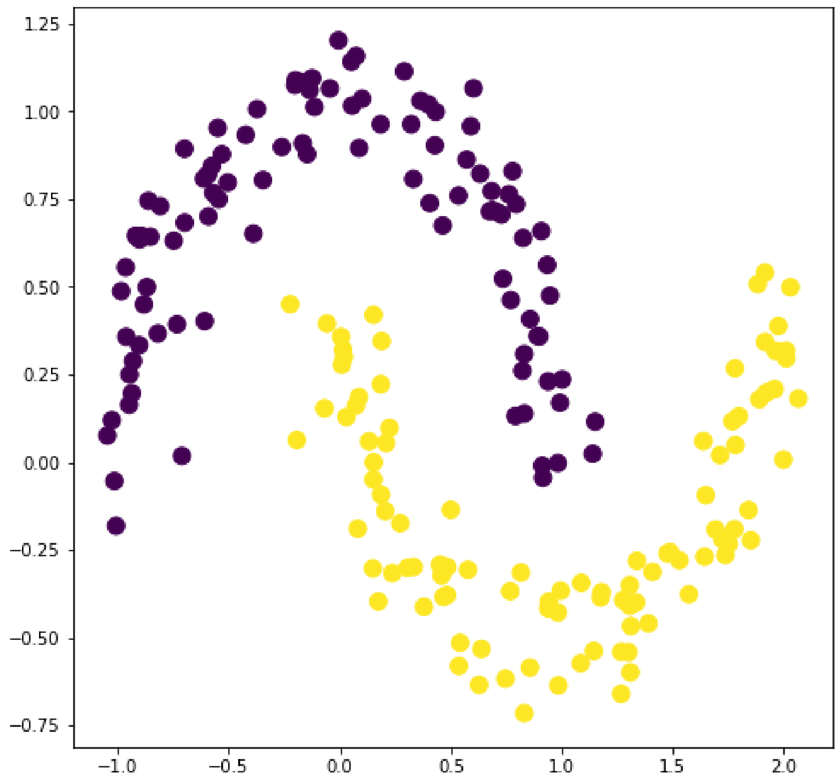
[54]:
```
eblobs = encoder.predict(blobs)
fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111)
plt.scatter(eblobs[:, 0], eblobs[:, 0], c=labels,s=100);
```



Nonlinear data

Now we will play with non linearly separable data. First the two moons dataset.

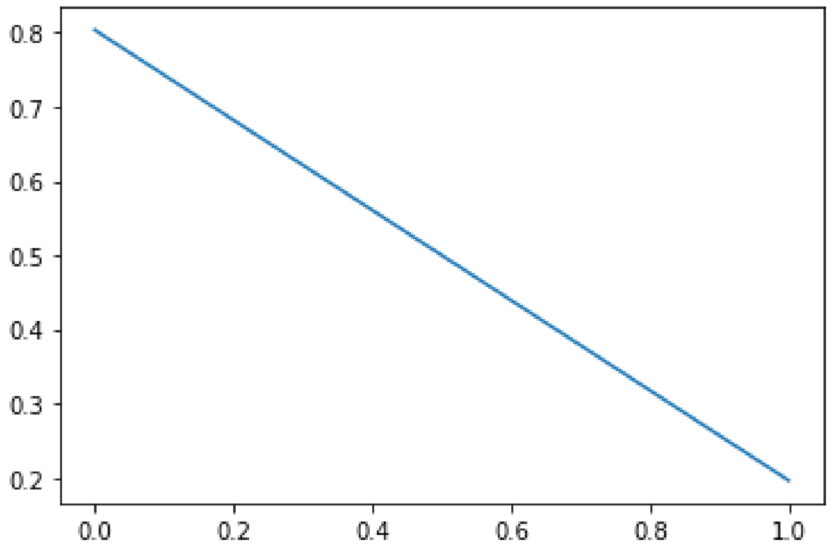
[55]:
```
moons, labelsm = make_moons(n_samples=200, noise=0.1)
fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111)
plt.scatter(moons[:, 0], moons[:, 1], c=labelsm, s=100);
```



PCA assigns most of the variance to the fist component, but the second one has still a significant amount

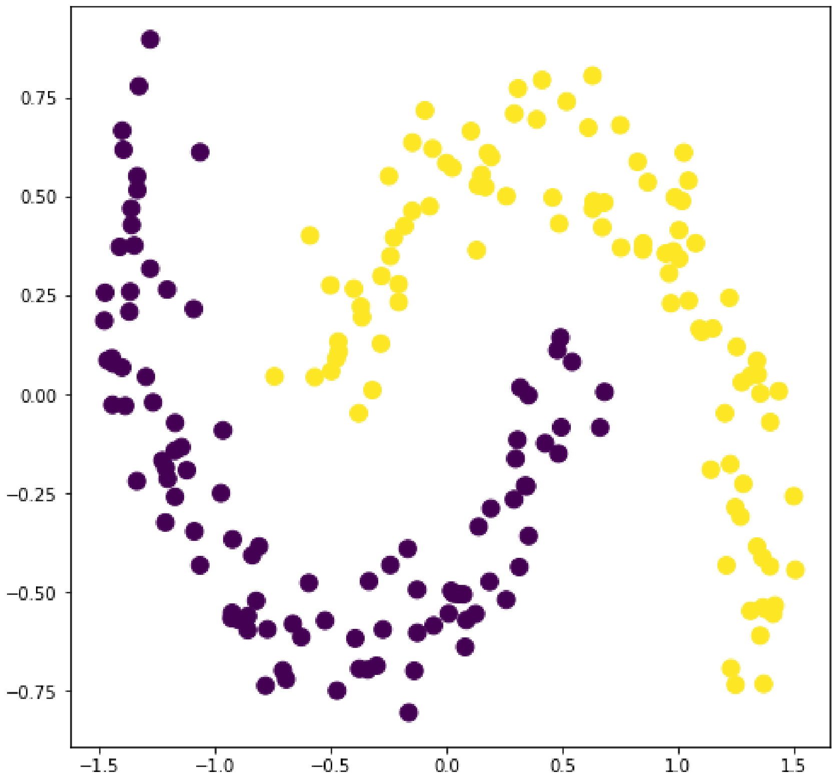
```
[56]: pca = PCA()
fdata = pca.fit_transform(moons)
print(pca.explained_variance_ratio_)
fig = plt.figure()
plt.plot(range(len(pca.explained_variance_ratio_)), pca.
  ↪explained_variance_ratio_);
```

```
[0.8035871 0.1964129]
```



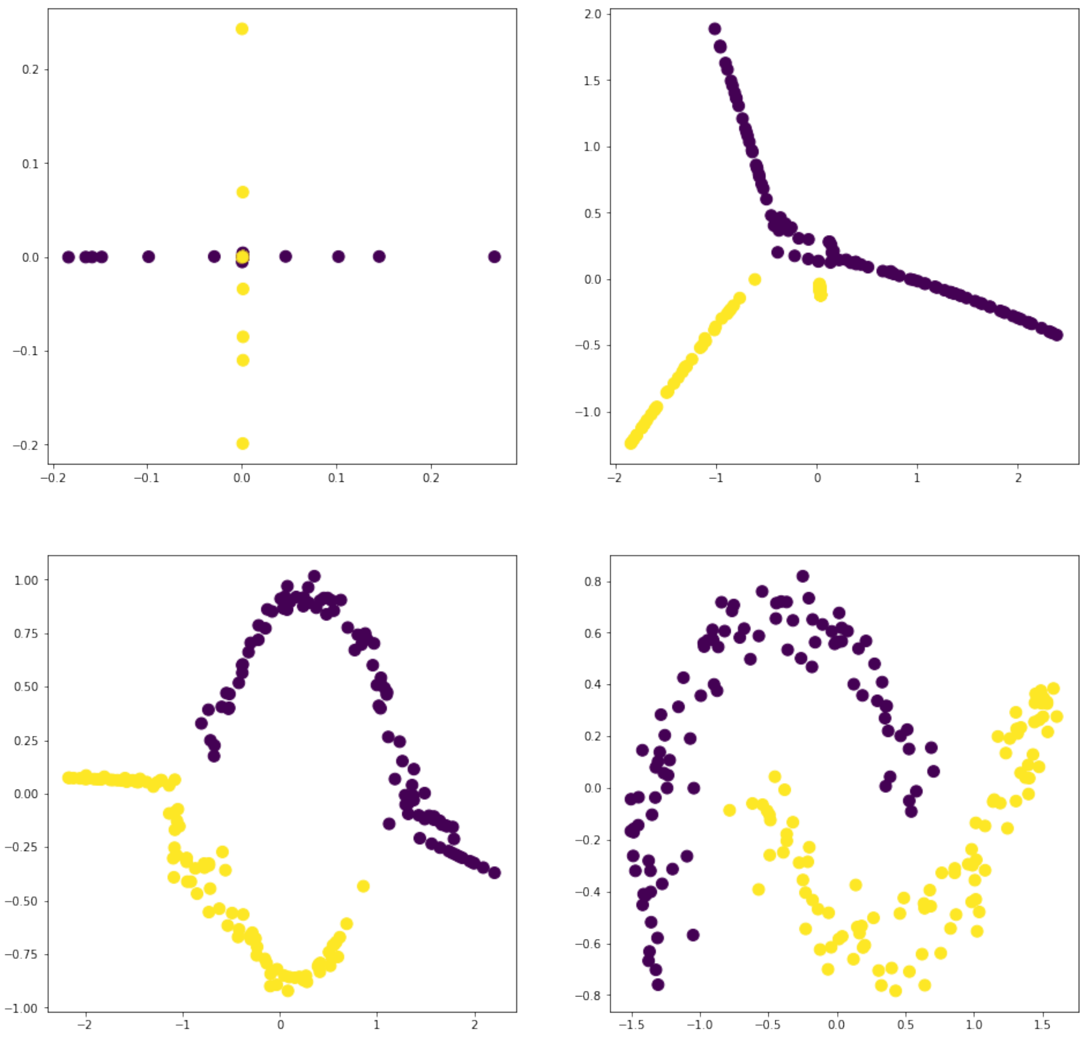
And PCA does not seems to have much effect on the data

[57]:
```
fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111)
plt.scatter(fdata[:, 0], fdata[:, 1], c=labelsm,s=100);
```



ISOMAP does a better job, but adjusting an adequate of neighbors to sparate the two moons requires some experimentation. Playing with the number of neighbors will results on different shapes that will converge on the original data for large values

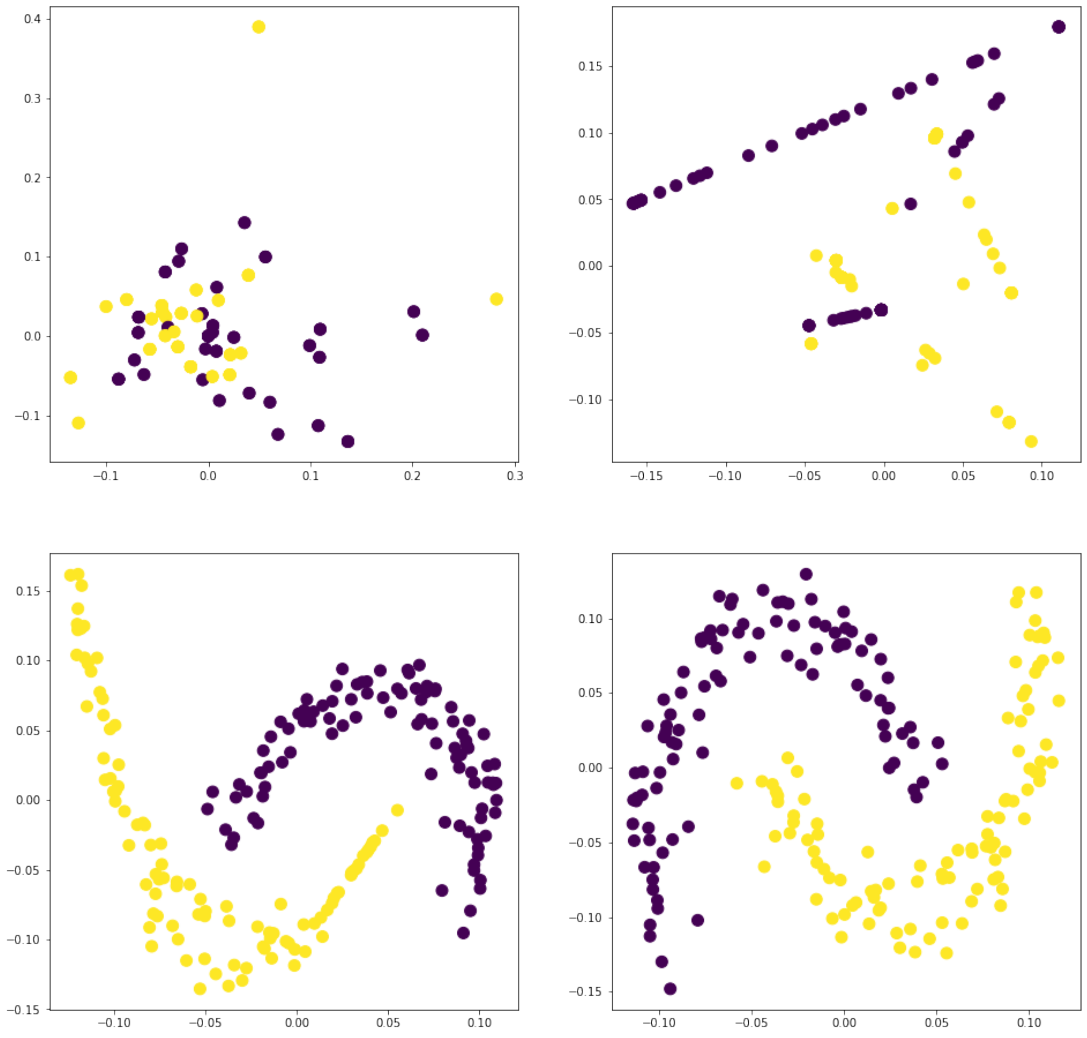
[58]:
```
nneigh=[1, 3, 10, 33]

fig = plt.figure(figsize=(16,16))
for i, nn in enumerate(nneigh):
    iso = Isomap(n_components=2, n_neighbors= nn)
    fdata = iso.fit_transform(moons)
    ax = fig.add_subplot(2,2,i+1)
    plt.scatter(fdata[:, 0], fdata[:, 1], c=labelsm, s=100);
```



LLE also allows separating both moons resulting in very different transformations depending on the number of neighbors.

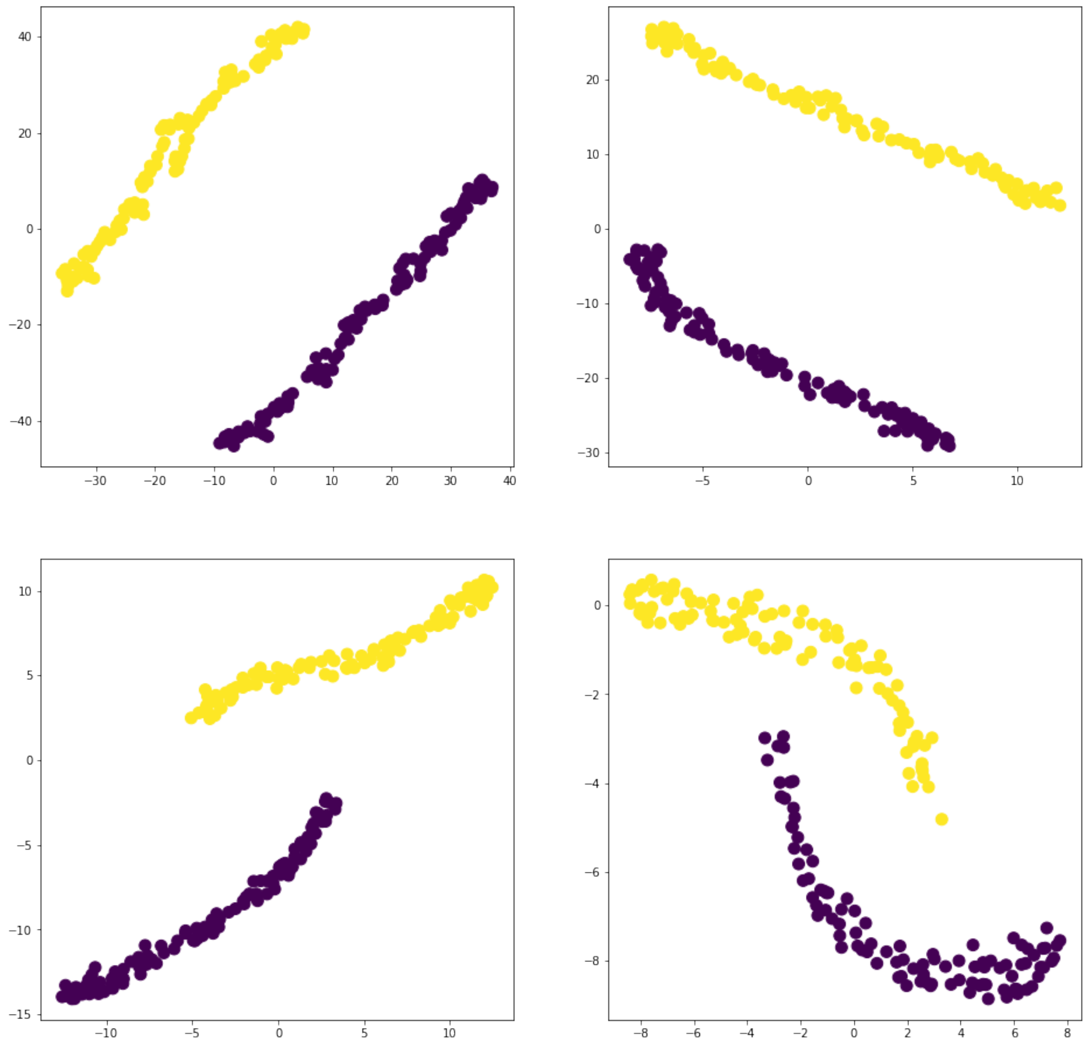
[59]:
```
nneigh=[1, 3, 10, 33]

fig = plt.figure(figsize=(16,16))
for i, nn in enumerate(nneigh):
    lle = LocallyLinearEmbedding(n_neighbors=nn, n_components=2,␣
 ↪method='standard', random_state=0)
    fdata = lle.fit_transform(moons)
    print (f'Reconstriction error {nn} = {lle.reconstruction_error_}')
    ax = fig.add_subplot(2,2,i+1)
    plt.scatter(fdata[:, 0], fdata[:, 1],  c=labelsm, s=100);
```

```
Reconstriction error 1 = -5.229028062269689e-15
Reconstriction error 3 = 1.5614311324968927e-15
Reconstriction error 10 = 6.331000012041501e-07
Reconstriction error 33 = 4.4980497323361285e-06
```



t-SNE is some times difficult to tune, but it can result in interesting representations

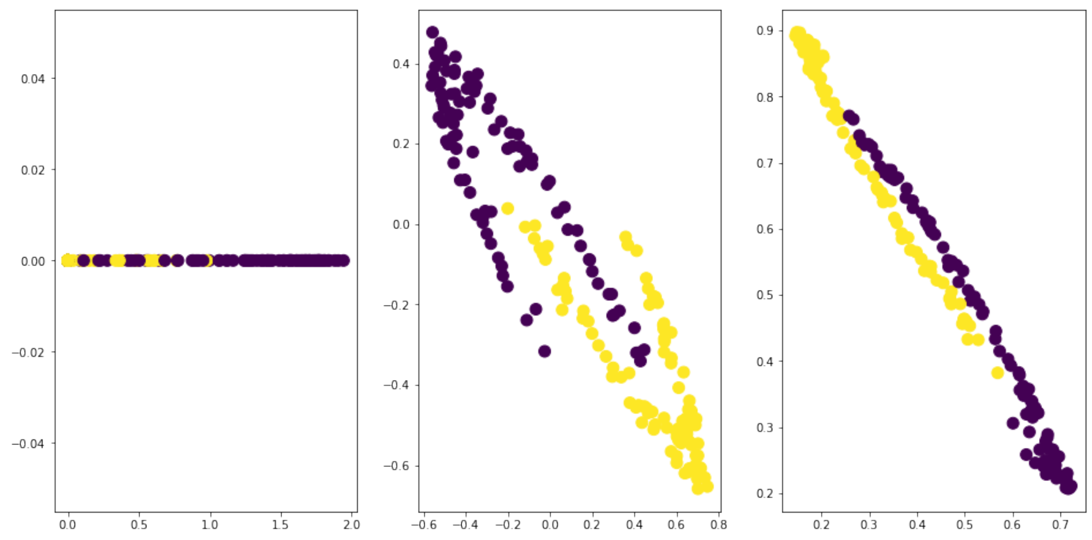
```
[60]: perp=[10, 20, 30, 50]

      fig = plt.figure(figsize=(16,16))
      for i, pp in enumerate(perp):
          tsne = TSNE(perplexity=pp, random_state=0)
          fdata = tsne.fit_transform(moons)
          ax = fig.add_subplot(2,2,i+1)
          plt.scatter(fdata[:, 0], fdata[:, 1],  c=labelsm, s=100);
```



We can also train an autoencoder with non linear activation functions so we obtain a non linear representation. In this case the input and the transformed spaces have the same number of dimensions so the results with a simple 2-2-2 architecture are not very good

[61]:
```
fig = plt.figure(figsize=(16,8))

for i, act in enumerate(['relu','tanh', 'sigmoid']):

    input = keras.Input(shape=(2,))
    encoded = layers.Dense(2, activation=act)(input)
    decoded = layers.Dense(2, activation='linear')(encoded)

    autoencoder = keras.Model(input, decoded)
    encoder = keras.Model(input, encoded)
    autoencoder.compile(optimizer='adam', loss='mse')
    autoencoder.fit(moons, moons, epochs=100, batch_size=32, shuffle=True,␣
 ↪verbose=False)

    emoons = encoder.predict(moons)

    ax = fig.add_subplot(1,3,i+1)
```

```
plt.scatter(emoons[:, 0], emoons[:, 1], c=labelsm,s=100);
```



1.4 Similarity/distance functions

Many unsupervised algorithms (like clustering algorithms) are based on the measure of the similarity/distance among examples. The values for this comparison are obtained applying these functions to the attribute representation of the examples. This means that we assume that there is a N-dimensional space where the examples are embedded and where such functions represent the relationships of the patterns in the data. There are domains where this assumption is not true so some other kind of functions will be needed to represent instances relationships.

Mathematically speaking, a function is a *similarity* if it holds the following properties:

1. $s(p,q) = 1 \iff p = q$
2. $\forall p,q\ s(p,q) = s(q,p)$ (symmetry)

Some examples of similarity function are:

- **Cosine similarity**

$$d(i,k) = \frac{x_i^T \cdot x_j}{\| x_i \| \cdot \| x_j \|} \tag{1.32}$$

  The similarity is computed as the cosine of the angle between two vectors, so there is a geometrical interpretation of the similarity.

- **Pearson correlation measure**

$$d(i,k) = \frac{(x_i - \overline{x}_i)^T \cdot (x_j - \overline{x}_j)}{\| x_i - \overline{x}_i \| \cdot \| x_j - \overline{x}_j \|} \tag{1.33}$$

  This distance corresponds to the cross correlation between two variables. The only difference with the cosine distance is that data is centered before computing the distance, so the result is the same if data is already centered.

On the other hand, a function is a *distance* if it holds the following properties:

1. $\forall p,q \;\; d(p,q) \geq 0 \;\; and \;\; \forall p,q \;\; d(p,q) = 0 \iff p = q$
2. $\forall p,q \;\; d(p,q) = d(q,p)$ (symmetry)
3. $\forall p,q,r \;\; d(p,r) \leq d(q,p) + d(p,r)$ (triangular inequality)

Distance functions can be unbounded, but in practice there is a maximum value defined by the ranges of the attributes that are used to describe the data. This can be used to normalize the values of the distances for example to a $[0,1]$ range. Some examples of distance functions are:

- **Minkowski metrics**

$$d(i,k) = \left(\sum_{j=1}^{d} |x_{ij} - x_{kj}|^{r} \right)^{\frac{1}{r}} \tag{1.34}$$

  The special cases of d=1 and d=2 are respectively the Manhattan (L_1) and Euclidean (L_2) distance functions that are the most common choices in practice.

- **Mahalanobis distance**

$$d(i,k) = (x_i - x_k)^T \cdot \varphi^{-1} \cdot (x_i - x_k) \tag{1.35}$$

  where φ is the covariance matrix of the attributes. This distance makes the assumption that attributes are not independent and this is represented by the covariance matrix. In the case of truly independent data this distance is the euclidean distance.

- **Chebyshev Distance**

$$d(i,k) = \max_{j} |x_{ij} - x_{kj}| \tag{1.36}$$

  It is also known as the maximum distance and it is also a Minkowski metric where $r = \infty$ (L_∞ metric).

- **Canberra distance**

$$d(i,k) = \sum_{j=1}^{d} \frac{|x_{ij} - x_{kj}|}{|x_{ij}| + |x_{kj}|} \tag{1.37}$$

  This is a weighted variation of the Manhattan distance.

Apart from the examples of similarity/distance functions presented, there are many more that are specific to different types of data or that are used in specific domains. For example in the case of *binary data* the Coincidence coefficient or the Jaccard coefficient are used among others, for *strings* a popular choice is the edit distance (also known as Levenstein distance) and its variations. For *temporal data* other distances that adapt to different time scales/speed like dynamic time warping (DTW) are more appropriate than for example the euclidean distance.

A very good reference on this topic is [29].

1.4.1 Notebook: Distance Functions

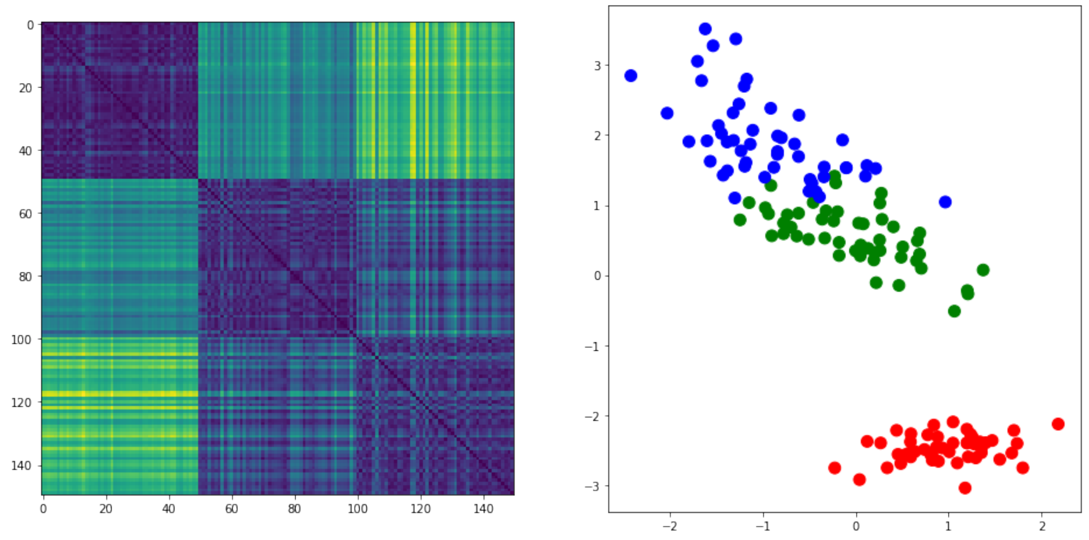
[1]:
```
from sklearn import datasets
from pylab import *
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import pairwise_distances
from sklearn.manifold import MDS
iris = datasets.load_iris()
col = ['r', 'g', 'b']
lc = [col[i] for i in iris['target']]
```

We can compute the euclidean distance for the data

$$d(x,y) = \sqrt{\sum_i (x_i - y_i)^2}$$

The plot on the left is the distance matrix, you can see the diagonal blocks that form the examples from each class, the plot on the right is the transformation of the distance matrix to a set of 2-D coordinates using Multidimensional Scaling

[2]:
```
mdist = pairwise_distances(iris['data'], metric='euclidean')
fig = plt.figure(figsize=(16,8))
fig.add_subplot(121)
plt.imshow(mdist)
mds = MDS(n_components=2,dissimilarity='precomputed', random_state=0)
fdata = mds.fit_transform(mdist)
fig.add_subplot(122)
plt.scatter(fdata[:, 0], fdata[:, 1], c=lc,s=100)
```



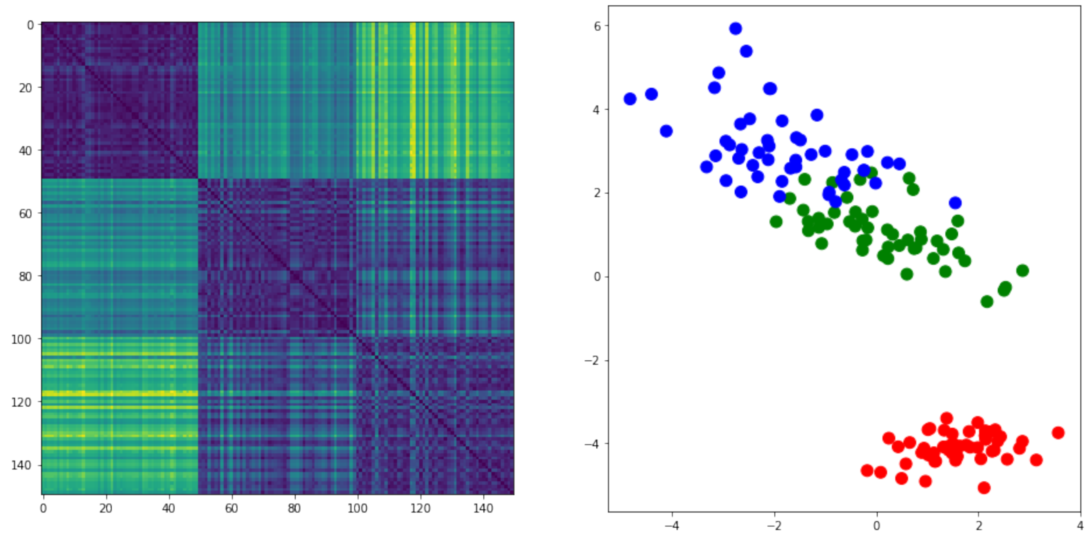
Changing the distance we change the space where the examples are embedded. We can do the same using the cityblock/manhattan distance:

$$d(x,y) = \sum_i |x_i - y_i|$$

The distances are similar, so the plot of the MDS coordinates is also similar.

[3]:
```
mdist = pairwise_distances(iris['data'], metric='cityblock')
fig = plt.figure(figsize=(16,8))
```

```
fig.add_subplot(121)
plt.imshow(mdist)
fdata = mds.fit_transform(mdist)
fig.add_subplot(122)
plt.scatter(fdata[:, 0], fdata[:, 1], c=lc,s=100)
```

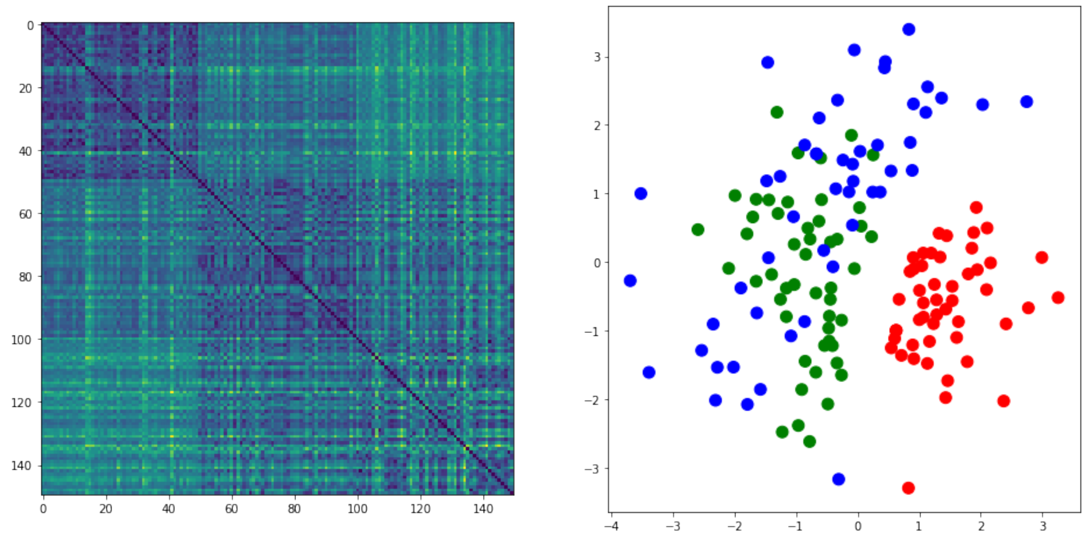


Mahalanobis distance is an euclidean distance that includes the covariance matrix, assuming that the attributes are not independent:

$$d(x,y) = \sqrt{(x-y)^T \Sigma^{-1} (x-y)}$$

This changes a lot the MDS transformation.

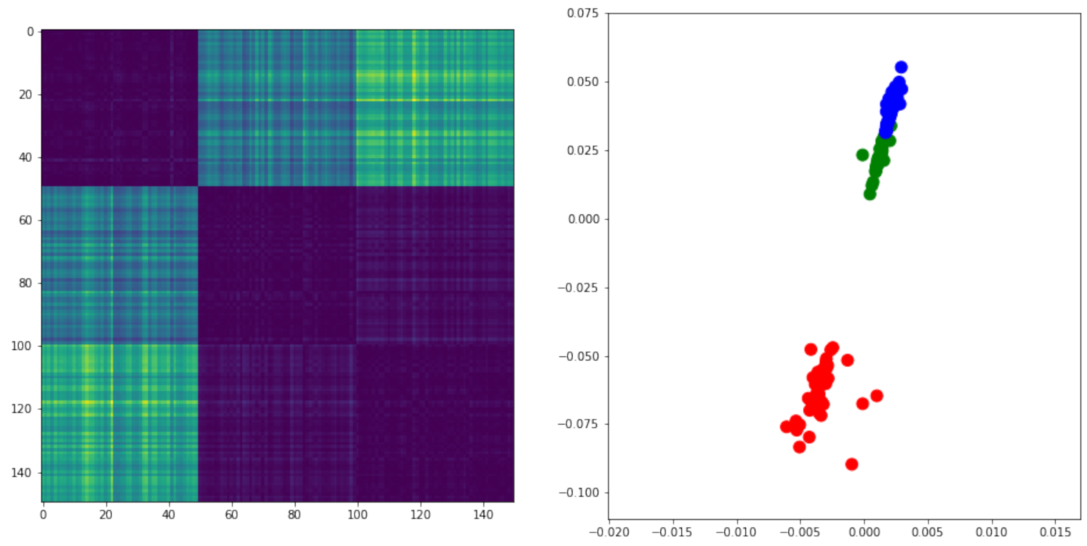
[4]:
```
mdist = pairwise_distances(iris['data'], metric='mahalanobis')
fig = plt.figure(figsize=(16,8))
fig.add_subplot(121)
plt.imshow(mdist)
fdata = mds.fit_transform(mdist)
fig.add_subplot(122)
plt.scatter(fdata[:, 0], fdata[:, 1], c=lc,s=100)
```



Cosine distance measures the angle of the vectors associated with the examples:

$$d(x,y) = \frac{X \cdot Y}{||X||\,||Y||}$$

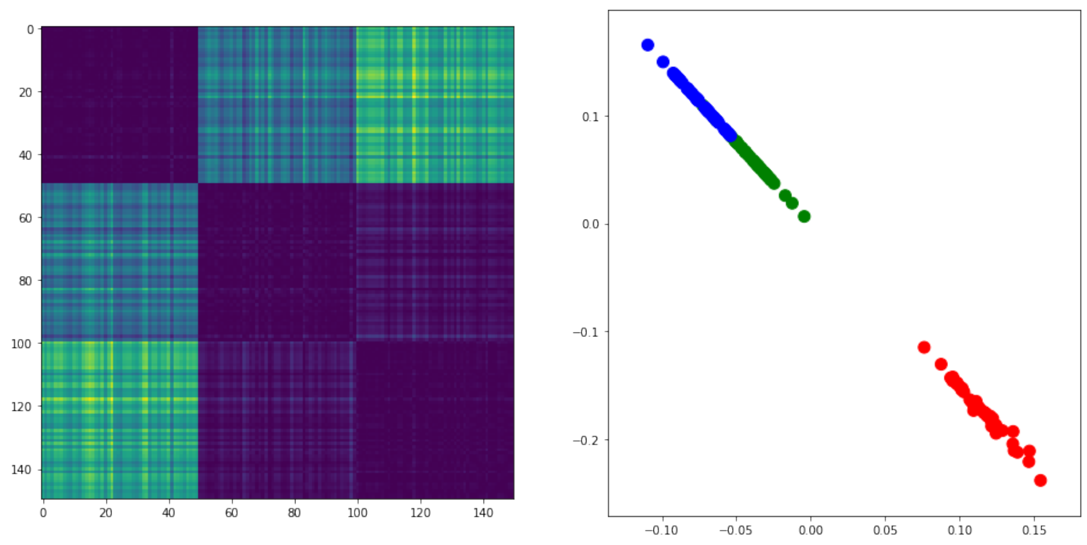
```
[5]: mdist = pairwise_distances(iris['data'], metric='cosine')
     fig = plt.figure(figsize=(16,8))
     fig.add_subplot(121)
     plt.imshow(mdist)

     fdata = mds.fit_transform(mdist)
     fig.add_subplot(122)
     plt.scatter(fdata[:, 0], fdata[:, 1], c=lc,s=100)
```



Correlation centers the data before computing the cosine distance

$$d(x,y) = \frac{(X-\overline{X}) \cdot (Y-\overline{Y})}{||(X-\overline{X})||\,||(Y-\overline{Y})||}$$

```
[6]: mdist = pairwise_distances(iris['data'], metric='correlation')
     fig = plt.figure(figsize=(16,8))
     fig.add_subplot(121)
     plt.imshow(mdist)

     fdata = mds.fit_transform(mdist)
     fig.add_subplot(122)
     plt.scatter(fdata[:, 0], fdata[:, 1], c=lc,s=100)
```



1.5 Examples

This section presents different examples of data preprocessing methods applied to real data extracted from the UCI ML repository and other practical applications.

1.5.1 Dimensionality reduction and discrete data

This example uses the mushroom dataset from the UCI ML repository. The data consist on 18 attributes describing qualitative characteristics of over 8000 mushrooms. This is a supervised dataset where the label classifies each mushroom as edible or poisonous.

The goal of this example is to visualize the data to obtain some insight about its structure. One problem with this dataset is that all the attributes are discrete, making it difficult to visualize the data. Figure 1.13 represents two of the dimensions for a subsample of the data, some jitter has been added to the graphic so overlapping points can be seen, otherwise only the $n \times m$ crossings of the values of the attributes would be visible.

The values of the attributes correspond to categories, so there is not an order that can be used to map the values to a continuous range. A possible transformation is to obtain binary attributes by *one-hot encoding*. This transformation generates as many binary attributes as values has the original attribute and each example is coded assigning a value 1 to the variable corresponding to the actual value and a 0 for the rest of attributes.

For example, one of the attributes of the dataset is the *gill-spacing* with values {`close`, `crowded`, `distant`}. The encoding will transform this variable into three variables (gill-spacing-close, gill-spacing-crowded, gill-spacing-distant) and for example the value `crowded` would be encoded as (gill-spacing-close=0, gill-spacing-crowded=1, gill-spacing-distant=0).

One has to be careful interpreting the results using this kind of transformations because the attributes obtained are not independent, so some assumptions of the dimensionality reduction methods would not hold.

Figure 1.14 presents the transformation of the one-hot encoding of the dataset using random projection, PCA and ISOMAP. As can be seen in the graphic, it seems to be more structure in the data than the one coded in the examples labels. The random projection transformation looks less evident, but despite that the projection only has two dimensions, each class seems to have some internal structure. The conditions for applying PCA to the data do not hold because attributes are not gaussian distributed, so the representation has to be cautiously interpreted.

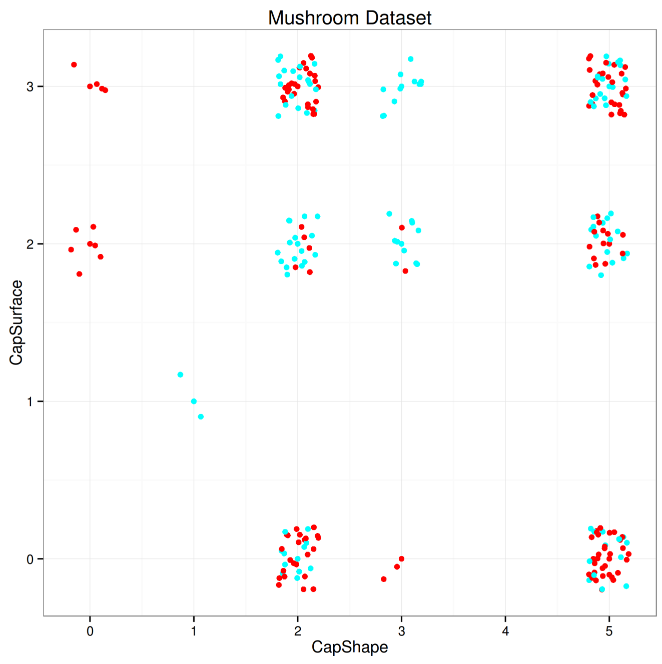


Figure 1.13: Mushroom data visualization for two attributes

Regardless, it is obvious that each class appears to be composed by several clusters indicating a more complex relationships in the data.

The ISOMAP transformation uses the euclidean distances among the examples, given that we are using one-hot encoding actually we are computing the hamming distance for the original nominal attributes. The number of neighbors used in the computations of ISOMAP has an influence on the results, so two values are presented. In the case of 100 neighbors the structure that shows the data is similar to the one obtained by PCA, using only 5 neighbors structure seems simpler, but there is also evidence of more information than the two labels used in the dataset.

1.6 Application: Wheelchair control

This example uses data collected from a research project about elderly patients with mobility problems. The goal is to study the patterns generated by the users of a wheelchair that has implemented a shared control mechanism. The idea is that the patient receives help from the chair in the case it detects that the movement is outside a set of quality parameters, mainly related to the avoidance of obstacles (walls, doors).

The wheelchair (see figure 1.15) is controlled by the patient using a joystick. The shared control mechanism is able to correct the input received in case it is needed. There is an array of laser sensors able to measure the distance from the front of the chair (210 degrees) to the nearest obstacle and an odometer for computing the position of the chair.

The dataset consists of collected trajectories of several patients in different predefined situations, like entering a room from a corridor through a door and then turning left to arrive to the goal point (see figure 1.16). There is a total of 88 attributes (see figure 1.17) consisting in the angle/distance to the goal, and the angle/distance measured by the sensors to the nearest obstacle around the chair.

The analysis goal is to characterize how the computer helps the patients with different

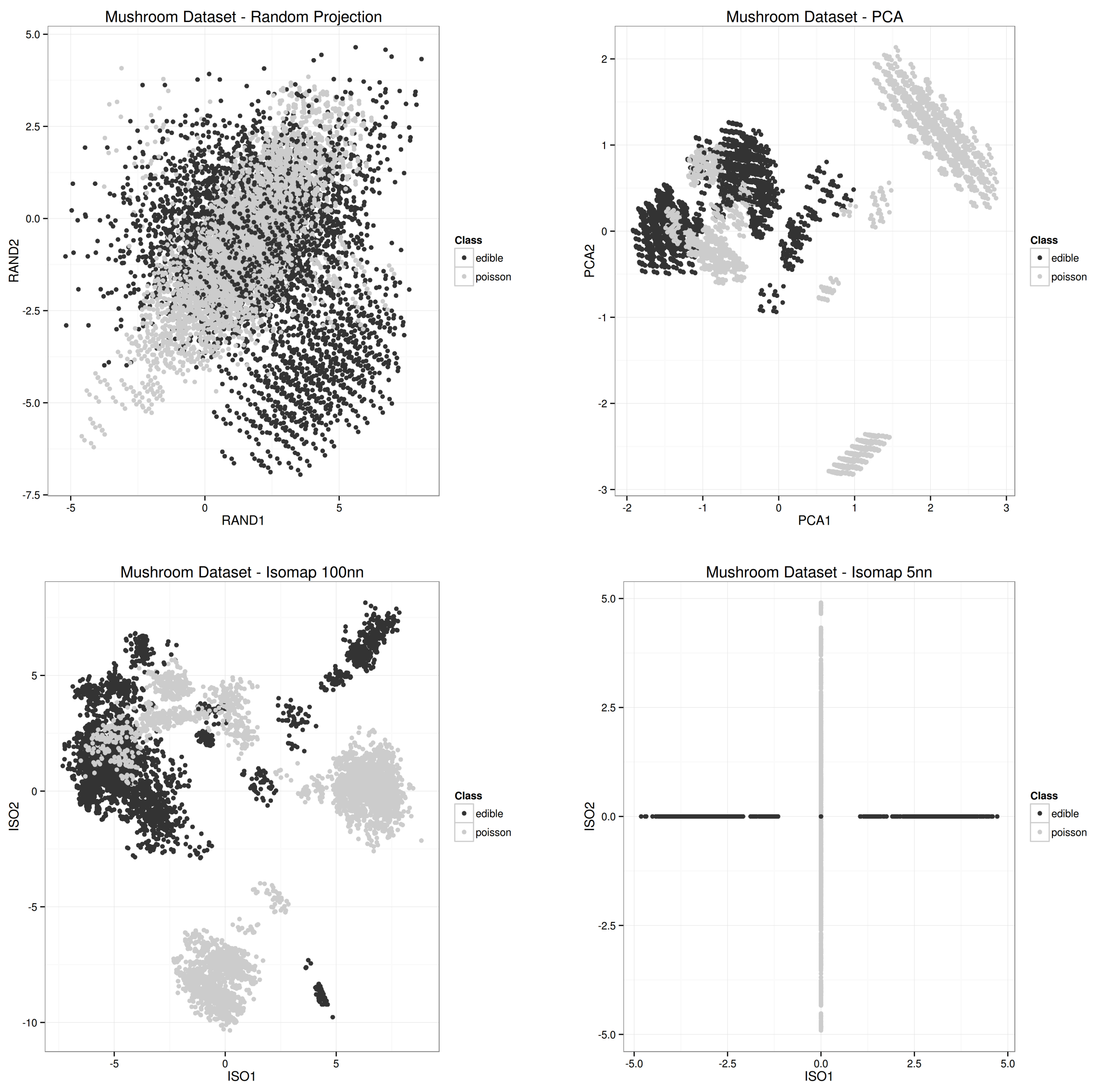


Figure 1.14: Mushroom data visualization after Random Projection, PCA and ISOMAP (100 and 5 neighbors) transformation

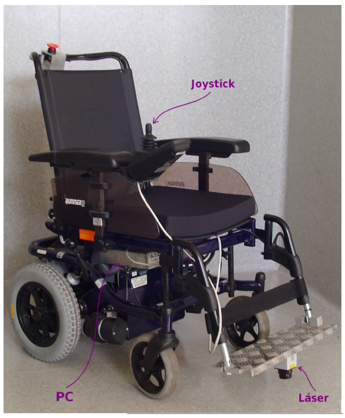


Figure 1.15: Wheel Chair with shared control

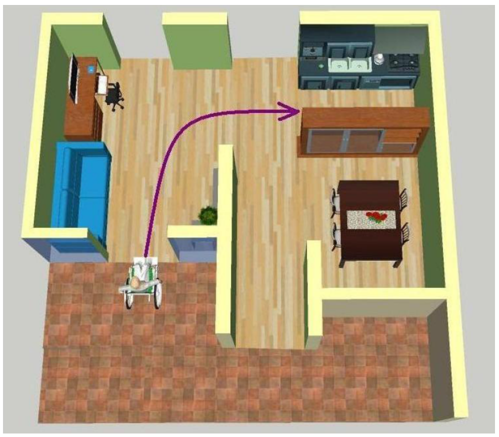


Figure 1.16: Crossing a door and left turn situation

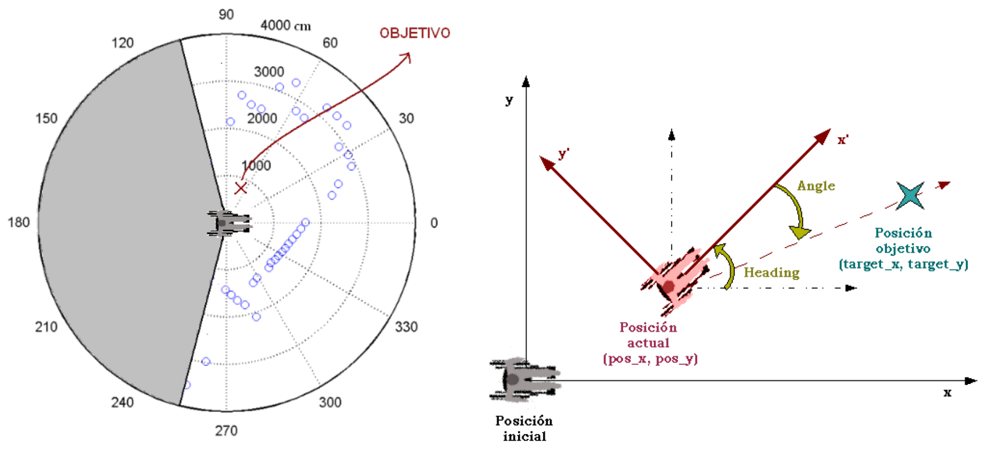


Figure 1.17: Attributes of the dataset

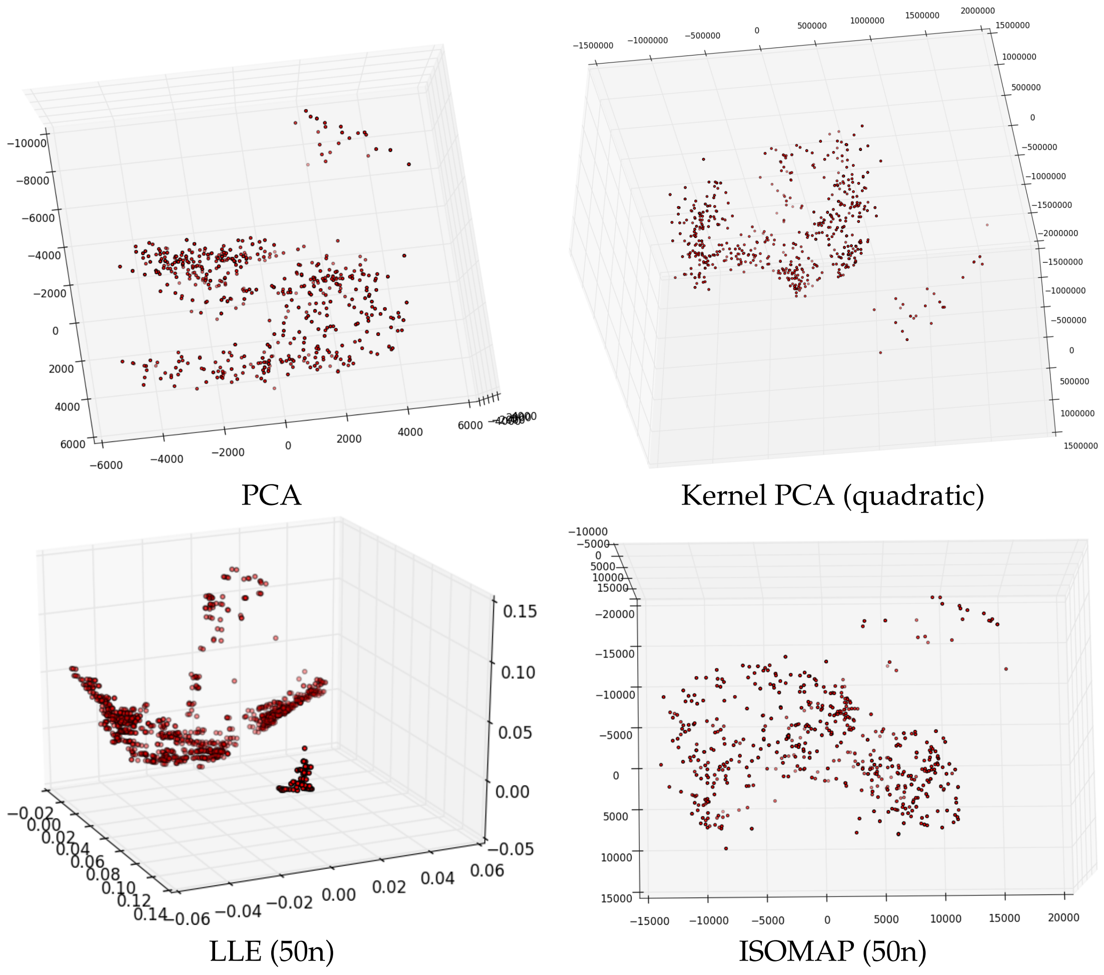


Figure 1.18: Wheelchair dataset visualization using PCA, Kernel PCA, Locally Linear Embedding (50n) and ISOMAP (50n)

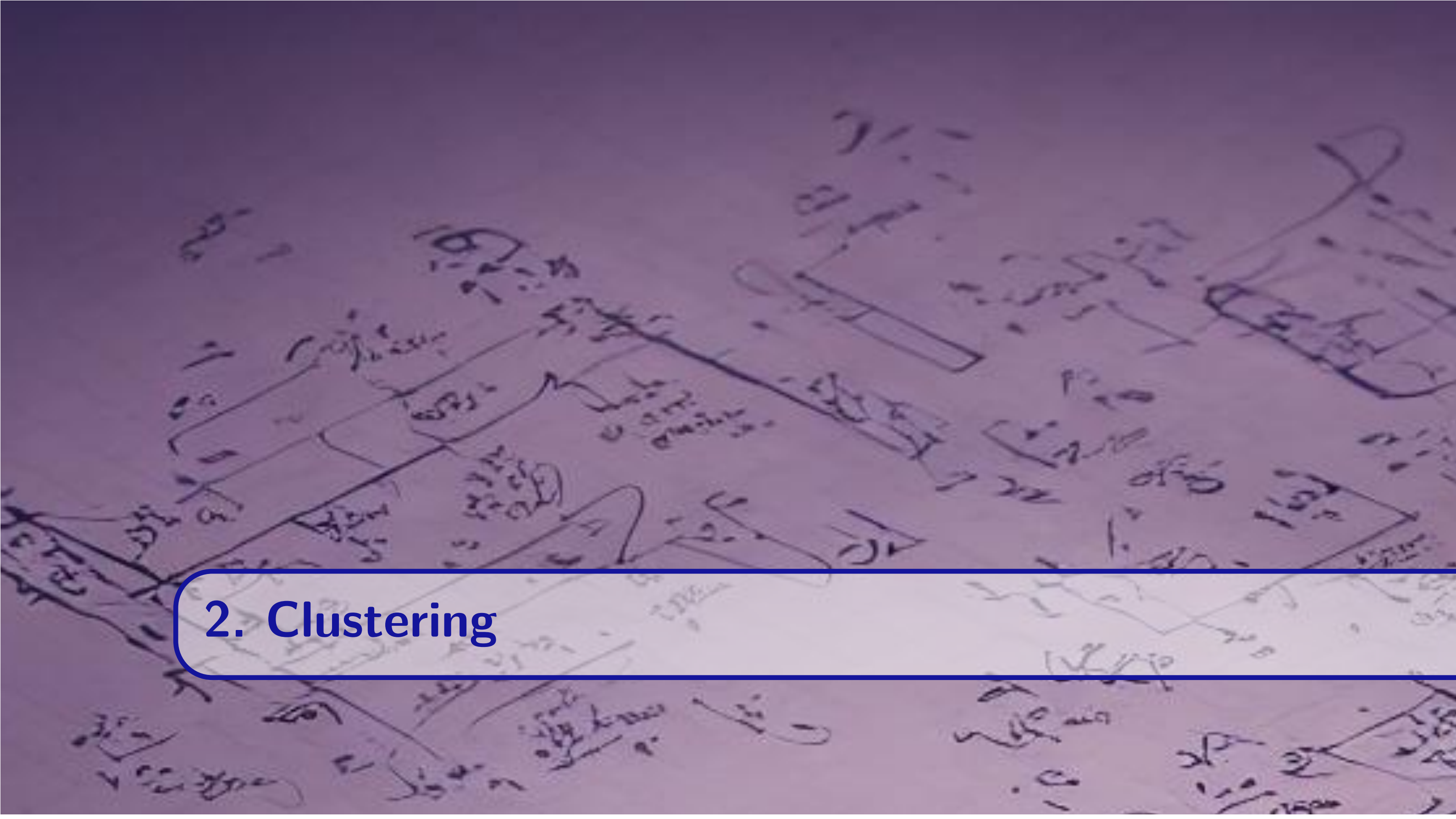
handicaps and to discover what patterns/structure appear in the trajectory data.

As in the previous example, we will test different dimensionality reduction methods to visualize the data and check for indications of structure. The first transformation using PCA (see figure 1.18) reveals several clusters in the data with elongated shapes, there is one cluster well separated and a density area that is difficult to appreciate, and that seems to have different densities.

This result gives the idea that probably a nonlinear transformation of the data will help to highlight the different density areas. A transformation using Kernel PCA with a quadratic kernel still maintains the separated cluster and obtains clusters with a more spherical shape. Other non lineal transformations reveal a similar structure. In this case LLE and ISOMAP have been computed with many neighbors. Usually this has the effect of obtaining a transformation closer to a linear transformation for these methods. This has the advantage of obtaining a representation that does not present strange shapes exaggerating small clusters and hidding larger clusters.

II

Clustering



2. Clustering

2.1 Clustering data

To learn/discover patterns from data can be done in a supervised or unsupervised way. The machine learning community has a strong bias towards supervised learning methods because the goal is well-defined, and the performance of a method given a dataset can be measured objectively. Although, it is obvious that we learn a lot of concepts unsupervisedly and that the discovery of new ideas is always unsupervised, so these methods are important from a knowledge discovery perspective.

Clustering can be described as an ill-posed problem in the sense that there is not a unique solution, because different clusterings of a dataset can be correct depending on the goals or perspective of the task at hand. The clustering task can be defined as a process that, using the intrinsic properties of a dataset $\mathcal{X}$, uncovers a set of partitions that represents its inherent structure. It is, thus, a task that relies on the patterns that arise from the values of the attributes that describe the dataset. The goal of the task will be either to obtain a representation that describes an unlabeled dataset (summarization) or to discover concepts inside the data (understanding).

Partitions can be either *nested*, so a hierarchical structure is extracted, or *flat*, no relation among the groups. The groups can also be *hard partitions*, so each example only belongs to one of the clusters, or *soft partitions*, so there is overlapping among them, that can be represented by a function measuring the membership of the examples to the clusters.

There are several approaches to obtain a partition of a dataset, depending on the characteristics of the data, or the kind of the desired partition. It is usual to assume that the data is embedded in an N-dimensional space that has a similarity/distance function defined. The main bias that clustering strategies apply is to assume that examples are more related to the nearest examples than to the farthest ones, and that clusters must be compact (dense) groups of examples that are maximally separated from other groups in the dataset. Broadly speaking, approaches that follow that bias can be divided in:

- *Hierarchical algorithms*, that result in a nested set of partitions representing the hierarchical structure of the data. The result is commonly a hard partition. These methods are usually based on a distance/similarity matrix of the examples and a recursive divisive or agglomerative strategy for the grouping of the data.

- *Partitional algorithms*, that result in a set of disjoint (hard) or overlapped (soft) partitions. They are based on the optimization of a criterion (assumptions about the characteristics of the cluster model). There is a more wide variety of methods of this kind, depending on the model assumed for the partition, or the grouping strategy used. The more representative ones include algorithms based on prototypes or probabilistic models, based on the discovery of dense regions, based on the subdivision of the space of features into a multidimensional grid and based on defining a graph structure among the examples.

In the following sections the main characteristics of these methods will be described with an outline of the main representative algorithms. As we will see, the methods for clustering data come from different areas like statistics, machine learning, graph theory, fuzzy sets theory or physics.

2.2 Statistical Hierarchical clustering

Hierarchical methods [39] use two strategies for building a tree of nested partitions that clusters a dataset, divisive and agglomerative. Divisive strategies begin with the entire dataset, and each iteration it is determined a way to separate the data into two partitions. This process is repeated recursively until individual examples are reached. Agglomerative strategies iteratively merge the most related pair of partitions according to a similarity/distance measure until there is only one partition. Usually agglomerative strategies are computationally more efficient.

These methods are based on a distance/similarity function that compares partitions and examples. Initially, these values for each pair of examples are stored in a matrix that the algorithm updates during the clustering process. This makes the computational cost of these algorithms at least $O(n^2d)$ in time and $O(n^2)$ in space before even starting the clustering.

The strategy used by some algorithms considers this matrix as a graph that is created by adding each iteration new edges in an ascending/descending order of length. In this case, a criterion determines when the addition of a new edge results in a new cluster. These are *graph based* algorithms. Other algorithms reduce the distance/similarity matrix each iteration by merging two groups, deleting these groups from the matrix and then adding the new merged group. These are *matrix algebra based* algorithms.

2.2.1 Graph based algorithms

These algorithms assume that the examples form a fully connected graph, the length of the edges is defined by the similarity/distance function and there is a connectivity criteria used to define when a new cluster is created. The most common ones are, the *single linkage* criteria, that defines a new cluster each time a new edge is added if it connects two disjoint groups, and the *complete linkage* criteria that considers that a new cluster appears only when the union of two disjoint groups forms a clique in the graph. Both criteria have different bias, single linkage allows discovering elongated clusters and complete link has preference for globular clusters.

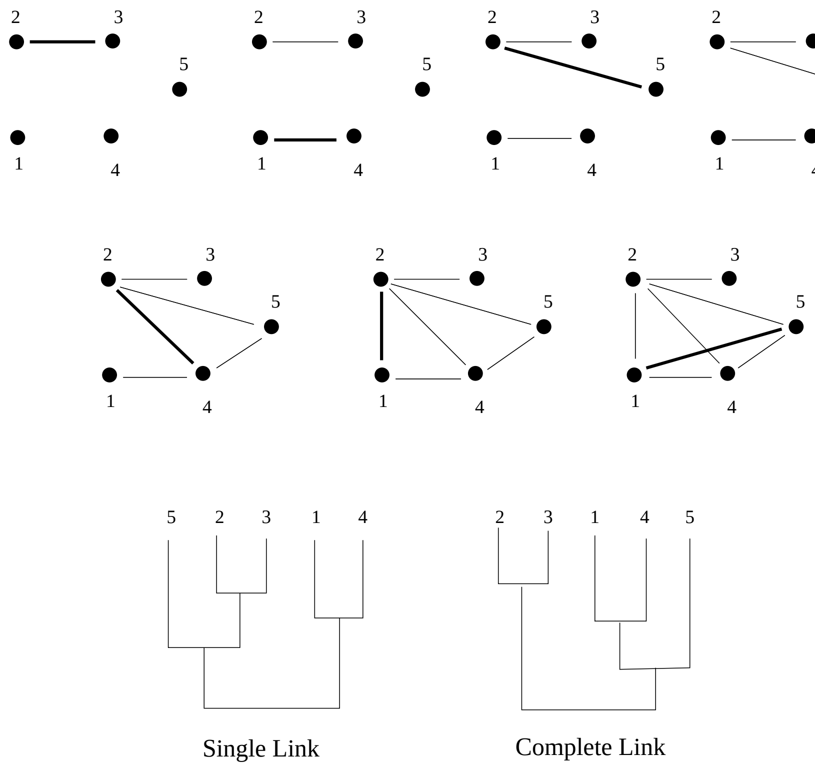
Algorithm 2.1 shows the agglomerative version of this clustering strategy, the criteria used to determine the creation of new clusters can be any of the mentioned ones. The cost of this algorithm is $O(n^3)$ being n the number of examples, it can be improved using a fast data structure to find the closest examples to $O(n^2 \log(n))$. It is also possible to assume that the graph is not fully connected to improve the computational cost, for instance computing the minimum spanning tree or with a graph of the k-nearest neighbors, but sacrificing the quality of the clustering.

Algorithm 2.1 Algorithm for graph agglomerative hierarchical clustering

```
Algorithm: Agglomerative graph algorithm
Compute distance/similarity matrix
repeat
    Find the pair of examples with the smallest distance
    Add an edge to the graph corresponding to this pair
    if Agglomeration criteria holds then
        Merge the clusters the pair belongs to
    end
until Only one cluster exists
```

As an example, lets consider the following matrix as the distances among a set of five data points:

	2	3	4	5
1	6	8	2	7
2		1	5	3
3			10	9
4				4



The algorithm begins adding a link between example 2 and 3 (the shortest distance), this would create the first cluster using the single link and complete link criteria (a clique of size 2). The next step adds the link between 1 and 4, that is a new cluster for both criteria. Third iteration adds the link between 2 and 5, that adds the example 5 to the cluster (2,3) for single link but does not create a new cluster for complete link because no new cliques have appeared. Having for single linkage already 2 clusters, the process can stop here, because there is only a last merge to perform, to know the distance of the merge, it is enough tho find the shortest link

Algorithm 2.2 Algorithm for matrix algebra based agglomerative hierarchical clustering

```
Algorithm: Agglomerative Clustering
Compute Distance/similarity matrix
repeat
    Find the pair of groups/examples with the smallest similarity
    Merge the pair of groups/examples
    Delete the rows and columns corresponding to the pair of groups/examples
    Add a new row and column with the new distances to the new group
until Matrix has one element
```

that connects a pair of examples from both clusters. For complete link, new links have to be added until a new clique is formed, this does not happen until the link between examples 1 and 5 appears. This means that the groups obtained using complete link in this case are different from the groups from single link. When only two partitions remain, the only clique that can join all the examples is the complete graph, to obtain the length of the least merge is enough to find the longest distance that links a pair of examples from both clusters.

2.2.2 Matrix algebra based algorithms

Matrix algebra based algorithms work similarly. The process initiates with the full distance matrix computed from the examples given a distance/similarity function. Each step of the process, it first finds the two most similar examples/clusters to merge into a new group, then a new distance is computed from the merged group to the remaining groups (see algorithm 2.2). Different updating criteria can be chosen for this.

The distances of this new group to the other groups is a combination of the distances to the two merged groups. Popular choices for this combination are the maximum, minimum and mean of these distances. The size of the group or its variance can be also used to weight the combination. Some of these choices replicate the criteria used by the graph based algorithms, for example single linkage is equivalent to substituting the distances by the minimum distance among the examples from the new group to the examples to the other groups, choosing the largest distance is equivalent to complete linkage.

This algorithm allows for a continuous variety of choices in the criteria for merging the partitions and updating the distances, this creates a continuum of algorithms that allows obtaining very different partitions from the same data. Most of the usual criteria can be described using the following formula:

$$d((i,j),k) = \alpha_i d(i,k) + \alpha_j d(j,k) + \beta d(i,j) + \gamma[d(i,k) - d(j,k) \tag{2.1}$$

For example, using $\alpha_i = \frac{1}{2}$, $\alpha_j = \frac{1}{2}$, $\beta = 0$ and $\gamma = -\frac{1}{2}$ we have the single linkage criteria. If we introduce the sizes of the groups as weighting factors, $\alpha_i = \frac{n_i}{n_i+n_j}$, $\alpha_j = \frac{n_j}{n_i+n_j}$ and $\beta = \gamma = 0$ computes the group average linkage. The Ward method that minimizes the variance of the groups corresponds to the weights $\alpha_i = \frac{n_i+n_k}{n_i+n_j+n_k}$, $\alpha_j = \frac{n_j+n_k}{n_i+n_j+n_k}$, $\beta = \frac{-n_k}{n_i+n_j+n_k}$ and $\gamma = 0$.

As an example, lets consider the same distance matrix as in the previous example:

	2	3	4	5
1	6	8	2	7
2		1	5	3
3			10	9
4				4

After merging the examples 2 and 3, and using the mean distance as criteria for computing the new distances we have:

	2,3	4	5
1	7	2	7
2,3		7.5	6
4			4

Now the closest groups are the examples 1 and 4, and after the distance recomputation:

	1,4	5
2,3	7.25	6
1,4		5.5

And finally the example 5 is the closest to the group (1,4).

	1,4,5
2,3	6.725

This finishes the clustering, the different mergings and their distances allows computing the dendrogram. This corresponds in this case to the dendrogram from the complete link criteria obtained in the example from the previous section.

2.2.3 Drawbacks

The main drawback of these algorithms is their computational cost. The distance matrix has to be stored and this scales quadratically with the number of examples. Also, the computational cost of these algorithms is cubic with the number of examples in the general case and this can be reduced in some particular cases to $O(n^2 \log(n))$ or even $O(n^2)$.

Also, the dendrogram as a representation of the relationship of the examples and clusters is only practical when the size of the dataset is small. This implies that the result of the algorithm has to be reduced to a partition instead of using the hierarchy. There are different criteria for obtaining a partition from the hierarchy that involve testing the gap in the distances among the cluster merges in the dendrogram or applying one of the several criteria for assessing cluster validity that will be described on the next chapter.

It is also a problem to decide the best linkage criterion. Each one assumes that different kinds of patterns exist in the data. For instance, single linkage is able to discover elongated and non-convex shaped clusters, but complete linkage or average linkage prefers globular clusters. Some criteria can also have issues when clusters of different sizes and densities appear in the data. This makes that the results using one or other criterion could be very different for the same dataset as can be seen in figure 2.1.

Some criteria present also some undesirable behaviors related to the behavior of the distance/similarity measure used to compare the examples. For example, *chaining effects* can usually appear when single linkage criterion is applied, this means that most or all of the merges consist in one individual example, and an existing cluster, so cutting the dendrogram yields a large cluster and several isolated examples. Also, the phenomenon known as *inversions* appear when the criterion considers the centroids of the clusters, this means that the distances in the dendrogram are not monotonically increasing with the height of the dendrogram, and two clusters merge at a distance that is lower than the previous merge. This can be observed clearly in figure 2.1 for the centroid linkage, but also appear in other dendrograms of the example.

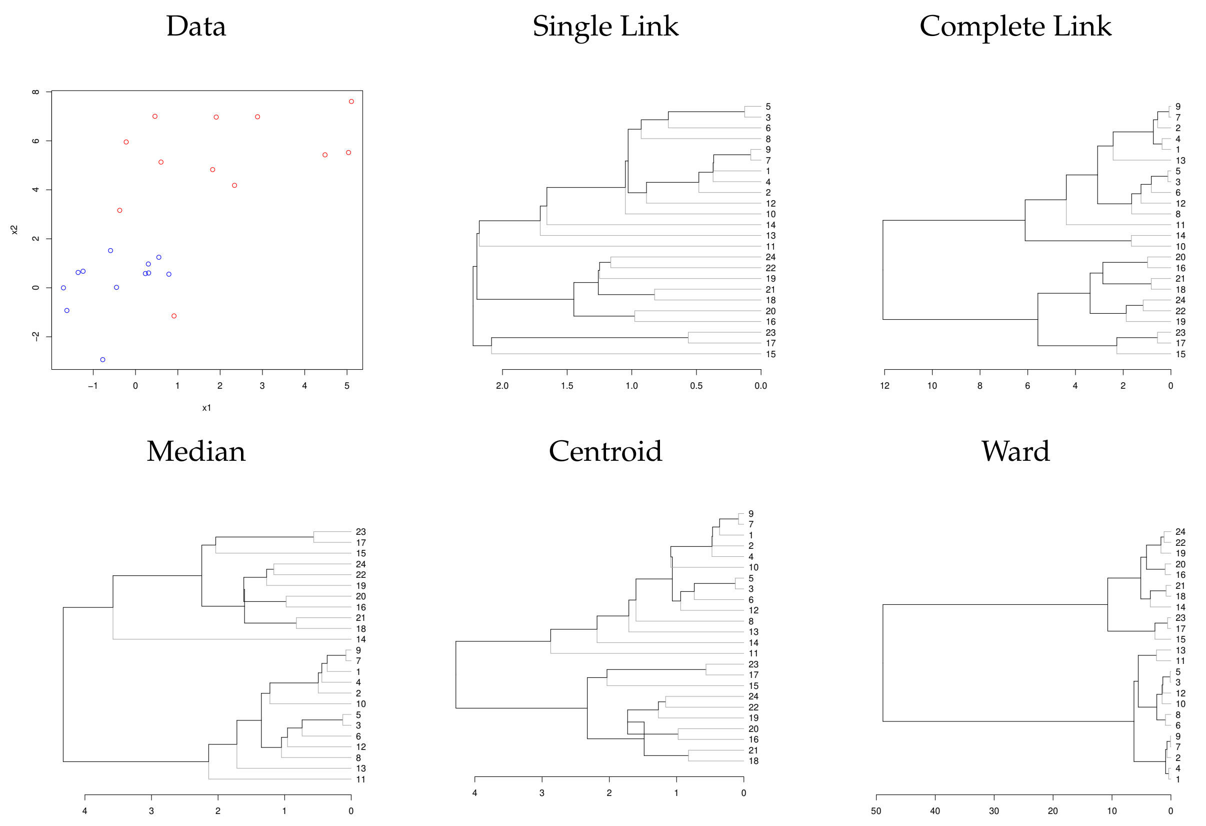


Figure 2.1: Dendrograms obtained applying different linkage criteria to the same dataset

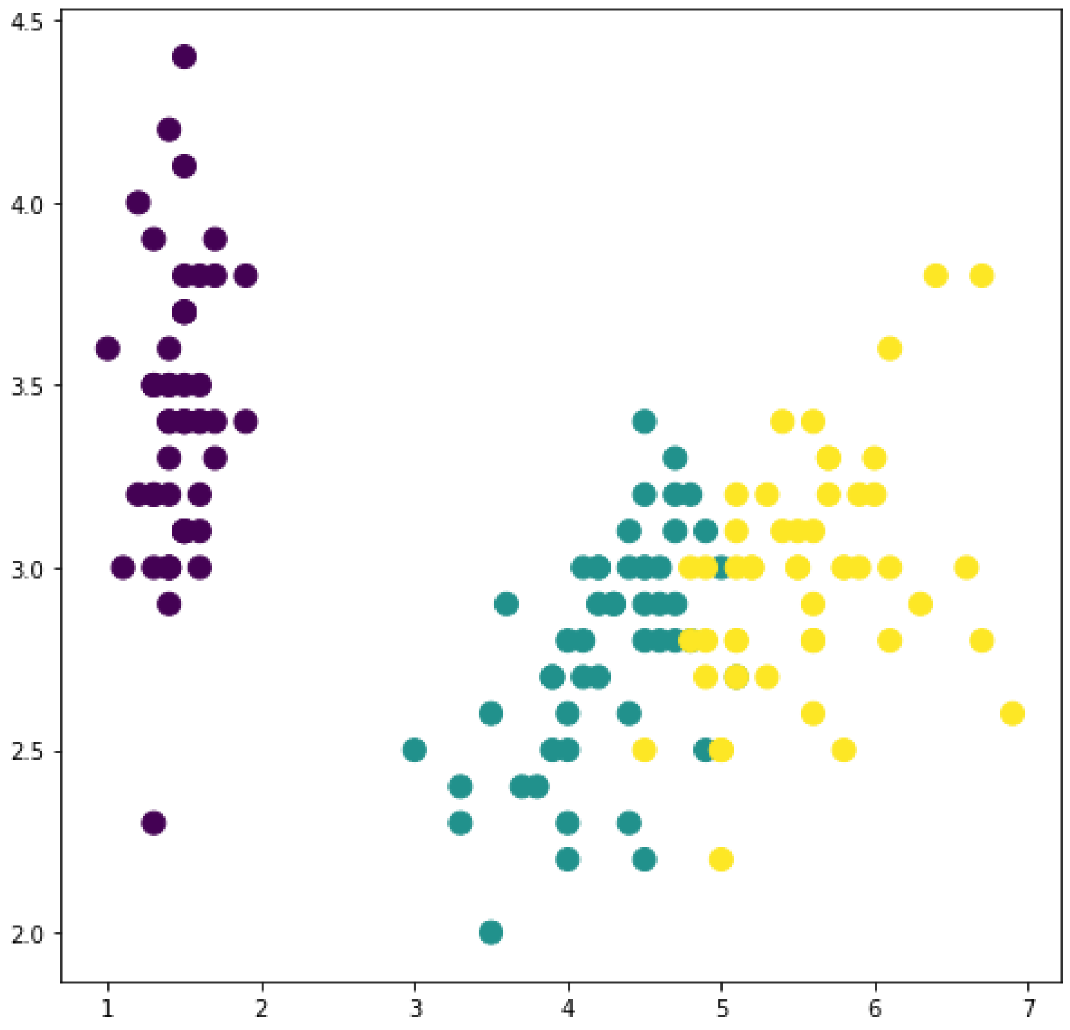
2.3 Notebook: Hierarchical Clustering

Now we are going to test the algorithms for hierarchical clustering using different aggregation criteria. We will use the implementation of these algorithms from the `scipy` library.

[1]:
```
from sklearn import datasets
from sklearn.metrics import adjusted_mutual_info_score
from scipy.cluster.hierarchy import linkage, dendrogram, fcluster
from pylab import *
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib notebook
```

We are going to use the functions that scipy provide for hierarchical agglomerative clustering. We will also continue working with the iris dataset.

[2]:
```
iris = datasets.load_iris()
plt.figure(figsize=(8,8))
plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=iris['target'],s=100);
```



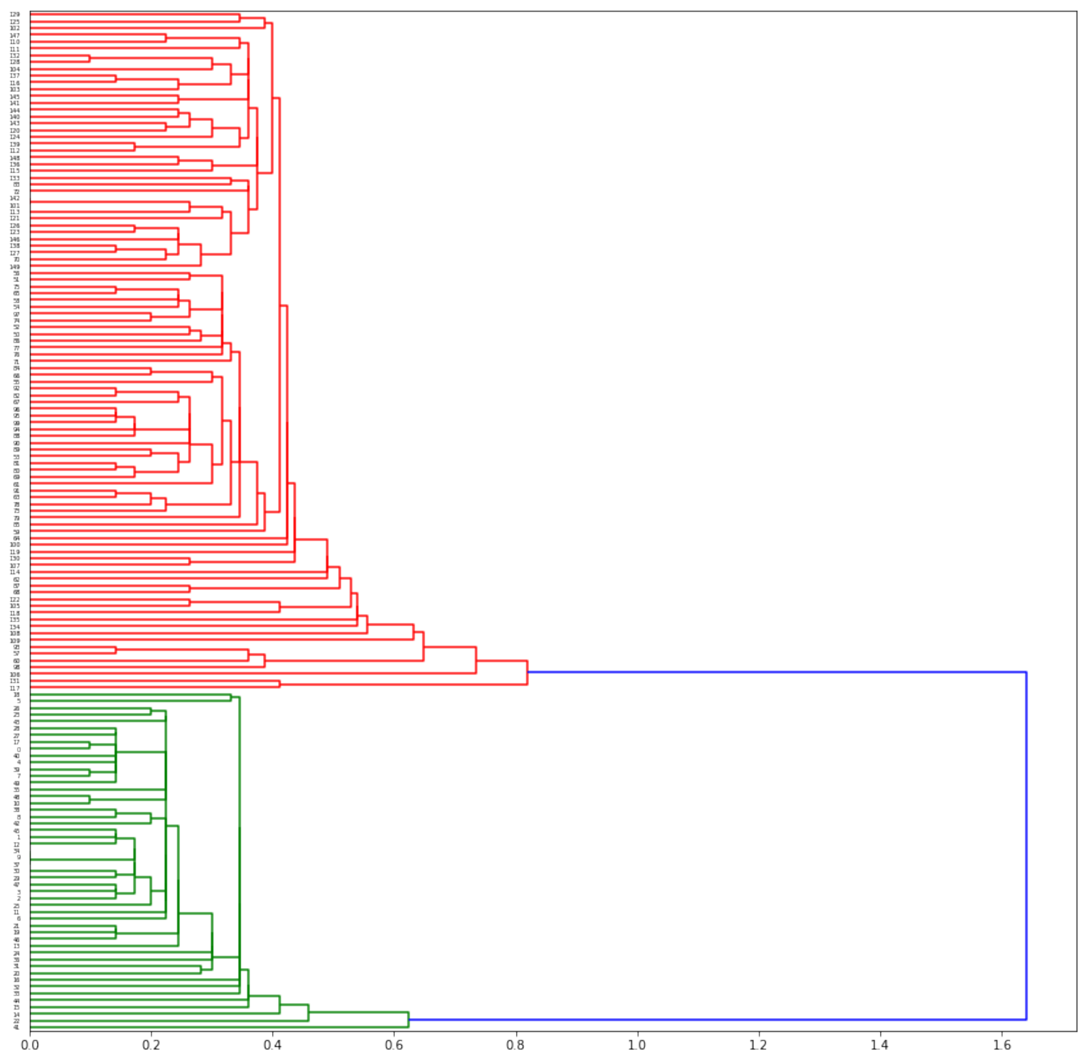
From the plot we can see that the classes from the labels do not form well separated clusters, so it is going to be difficult for hierarchical clustering to discover these three clusters.

First we apply **single linkage** clustering to the iris dataset.

Take note of the running time, so we can compare it with the running time of the other algorithms.

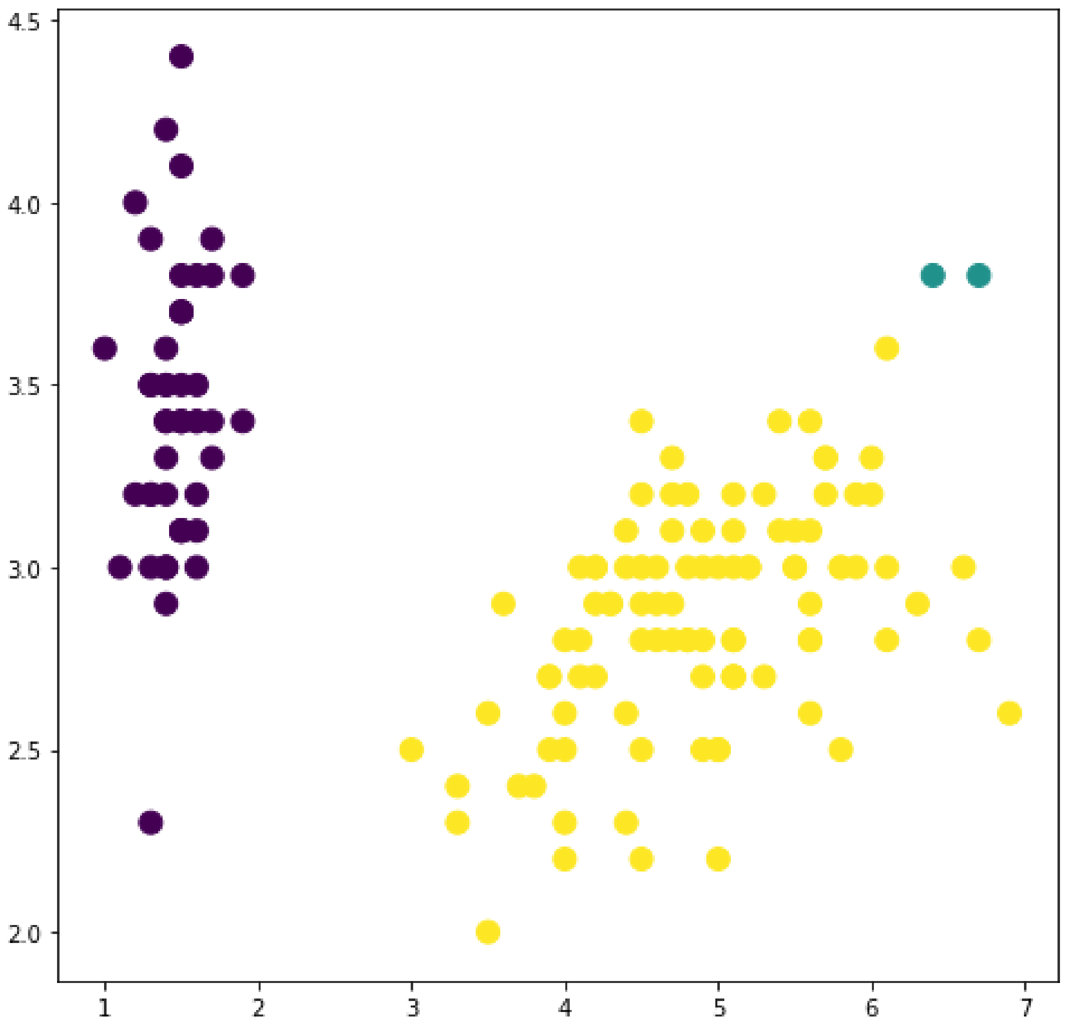
[3]:
```
%time clust = linkage(iris['data'], method='single')
plt.figure(figsize=(15,15))
dendrogram(clust, distance_sort=True, orientation='right');
```

```
CPU times: user 5.79 ms, sys: 6.9 ms, total: 12.7 ms
Wall time: 6.34 ms
```



There is only evidence of two distinctive partitions in the dataset, some inversions also appear on the dendrogram. If we cut the dendrogram so we have tree clusters we obtain the following

[4]:
```
plt.figure(figsize=(8,8))
clabels = fcluster(clust, 3, criterion='maxclust')

plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=clabels ,s=100);
```



We can compare the true labels with the ones obtained using this clustering algorithm using for example the mutual information score

```
[5]: print("AMI= ", adjusted_mutual_info_score(iris['target'], clabels))
```

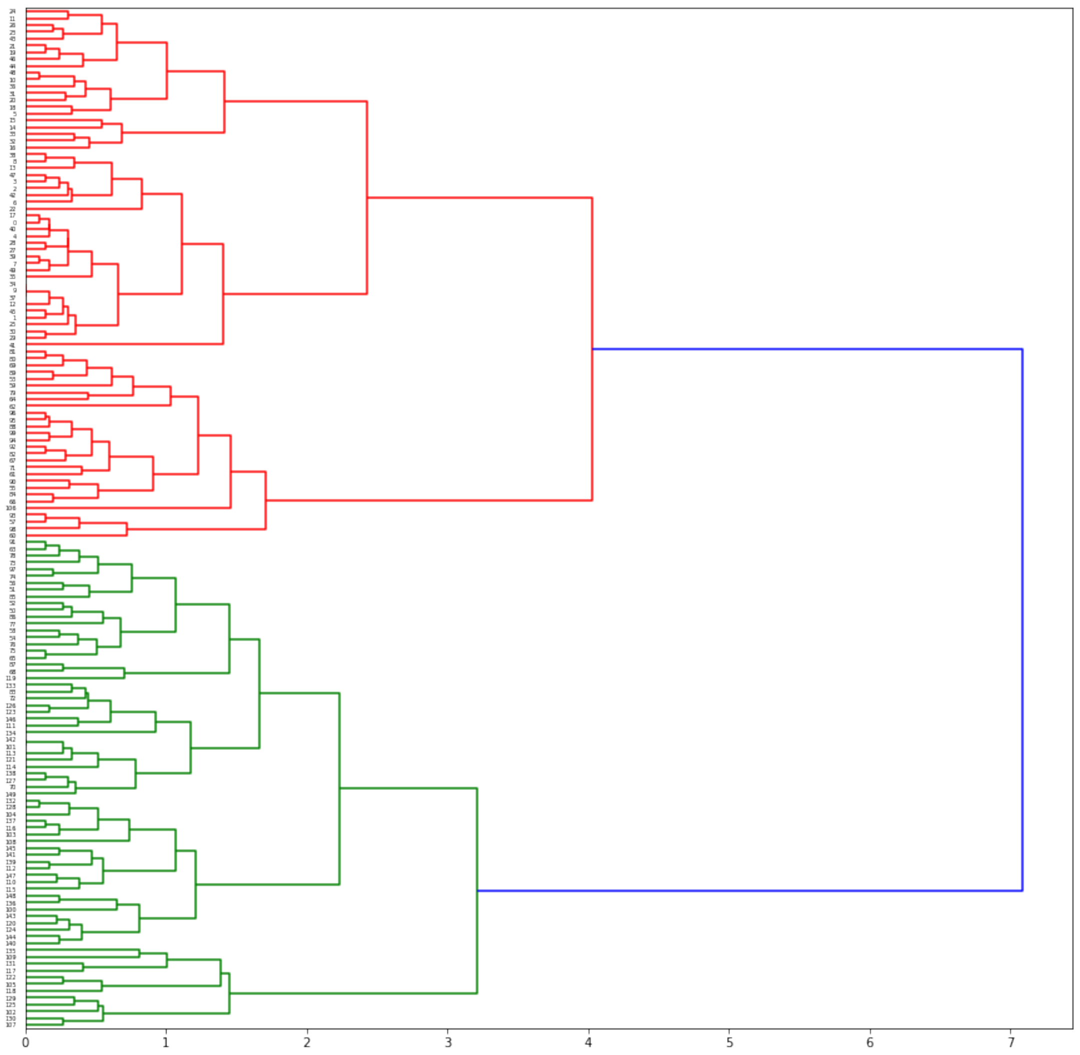
```
AMI=  0.5820928222202184
```

Not a very good result

Lets apply the **complete link** criteria to the data.

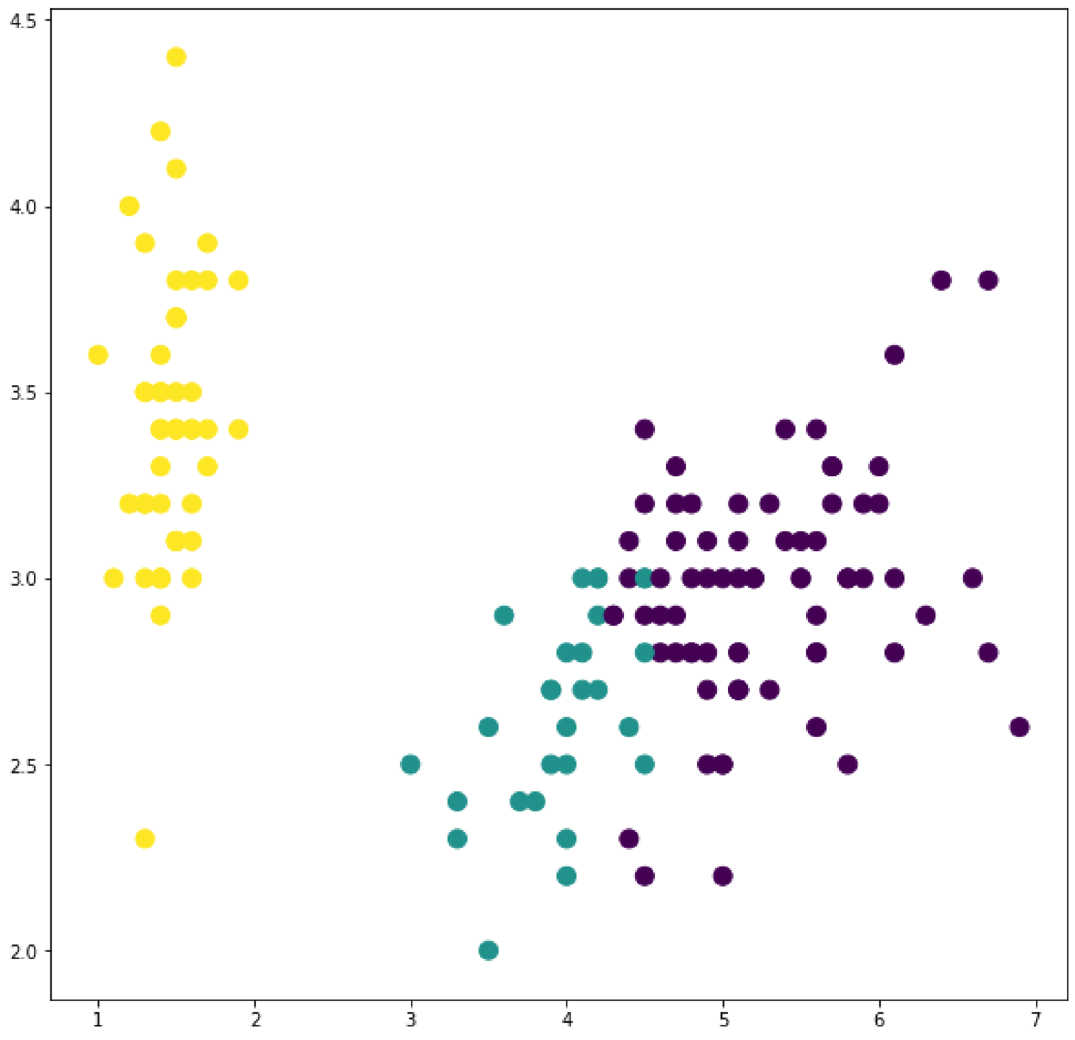
```
[6]: %time clust = linkage(iris['data'], method='complete')
     plt.figure(figsize=(15,15))
     dendrogram(clust, distance_sort=True, orientation='right');
```

```
CPU times: user 1.72 ms, sys: 298 µs, total: 2.02 ms
Wall time: 1.62 ms
```



Also two aparent clusters, but if we cut the dendrogram to three clusters we obtain something a little better.

[7]:
```
plt.figure(figsize=(10,10))
clabels = fcluster(clust, 3, criterion='maxclust')
plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=clabels,s=100);
```



If we compute the mutual information score

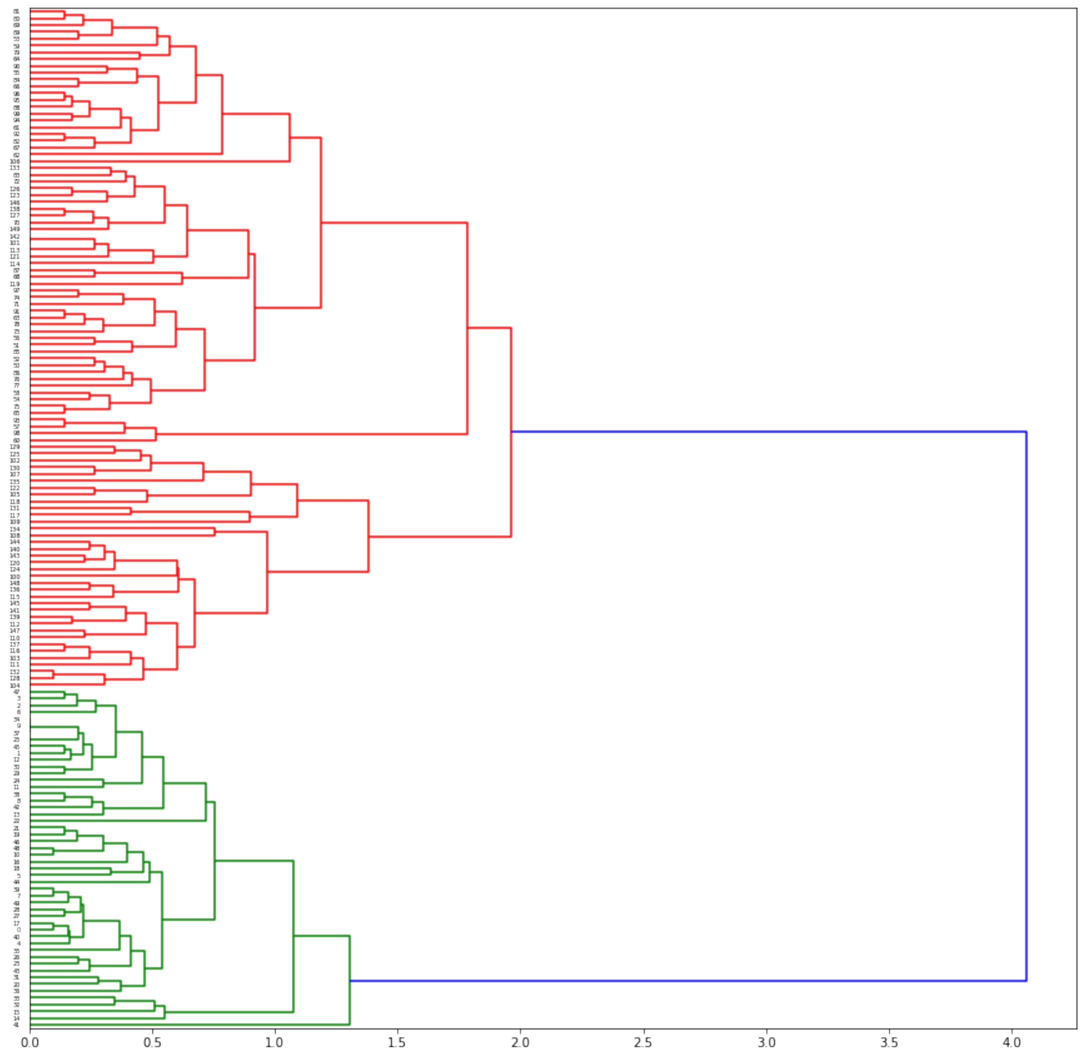
[8]:
```
print (adjusted_mutual_info_score(iris['target'], clabels))
```

```
0.6963483696671463
```

Now we use the **average link** criteria to the data.

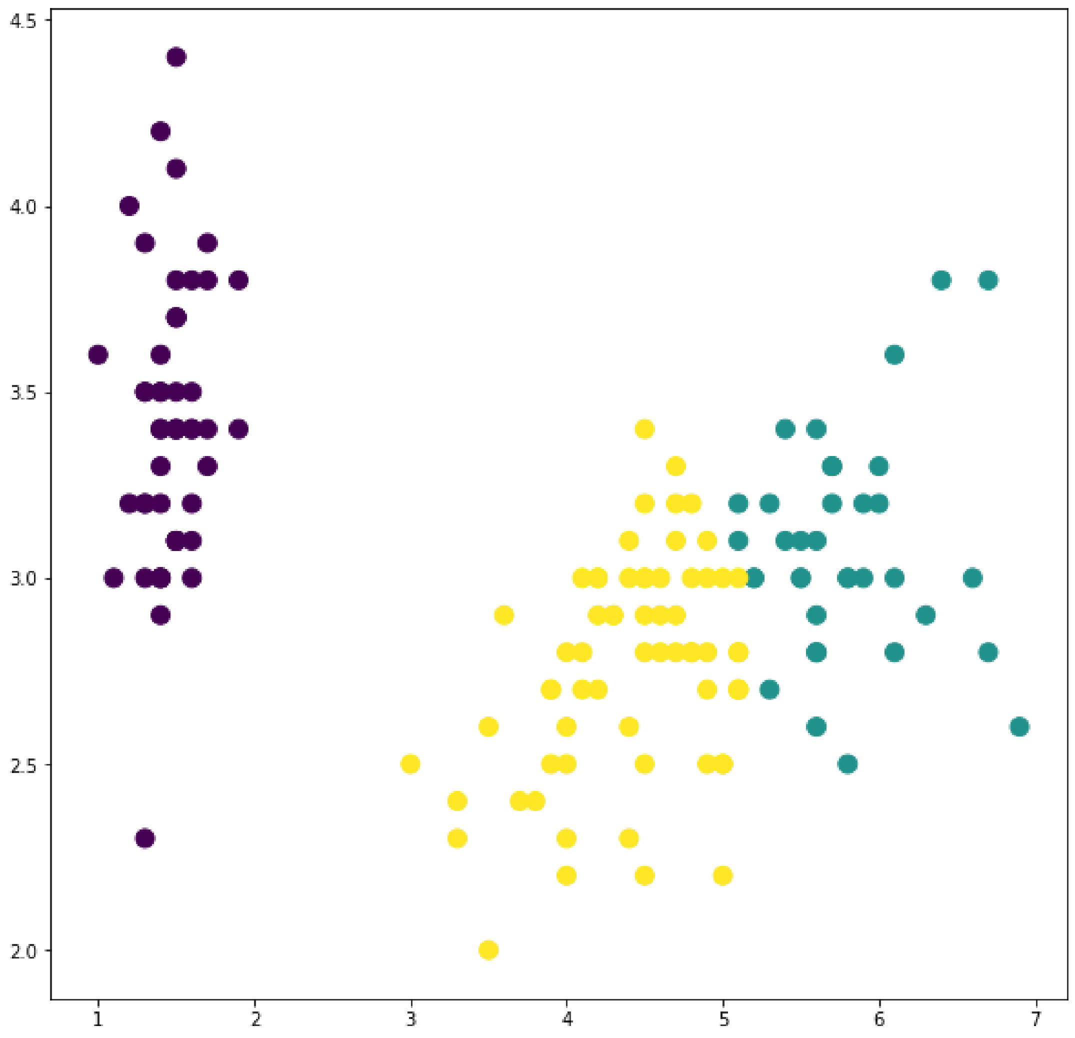
[9]:
```
%time clust = linkage(iris['data'], method='average')
plt.figure(figsize=(15,15))
dendrogram(clust, distance_sort=True, orientation='right');
```

```
CPU times: user 442 µs, sys: 1.76 ms, total: 2.2 ms
Wall time: 746 µs
```



We cut again to obtain three classes

```
[10]: plt.figure(figsize=(10,10))
      clabels = fcluster(clust, 3, criterion='maxclust')
      plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=clabels,s=100);
```



In this case the mutual information scores higher for this criteria.

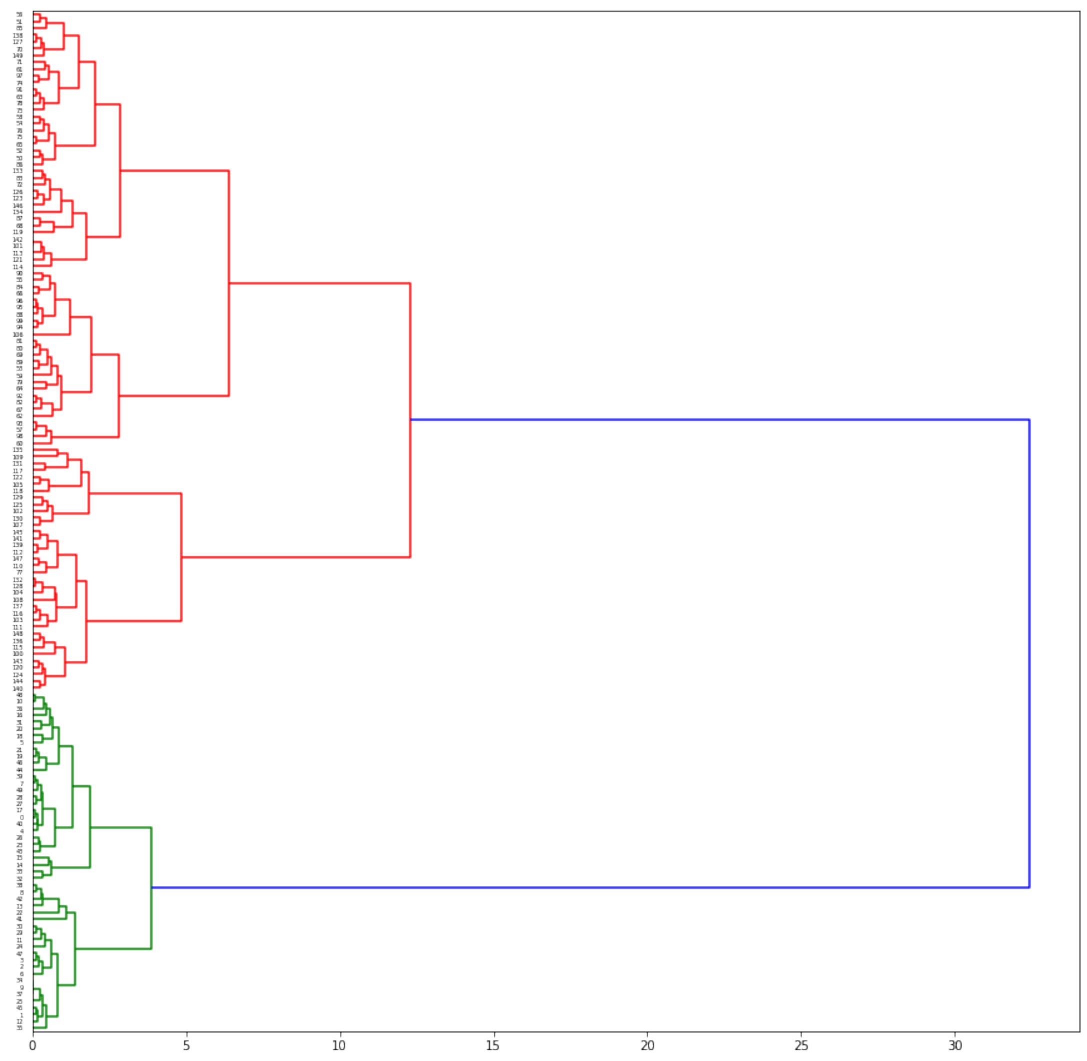
```
[11]: print (adjusted_mutual_info_score(iris['target'], clabels))
```

```
0.7934250515435666
```

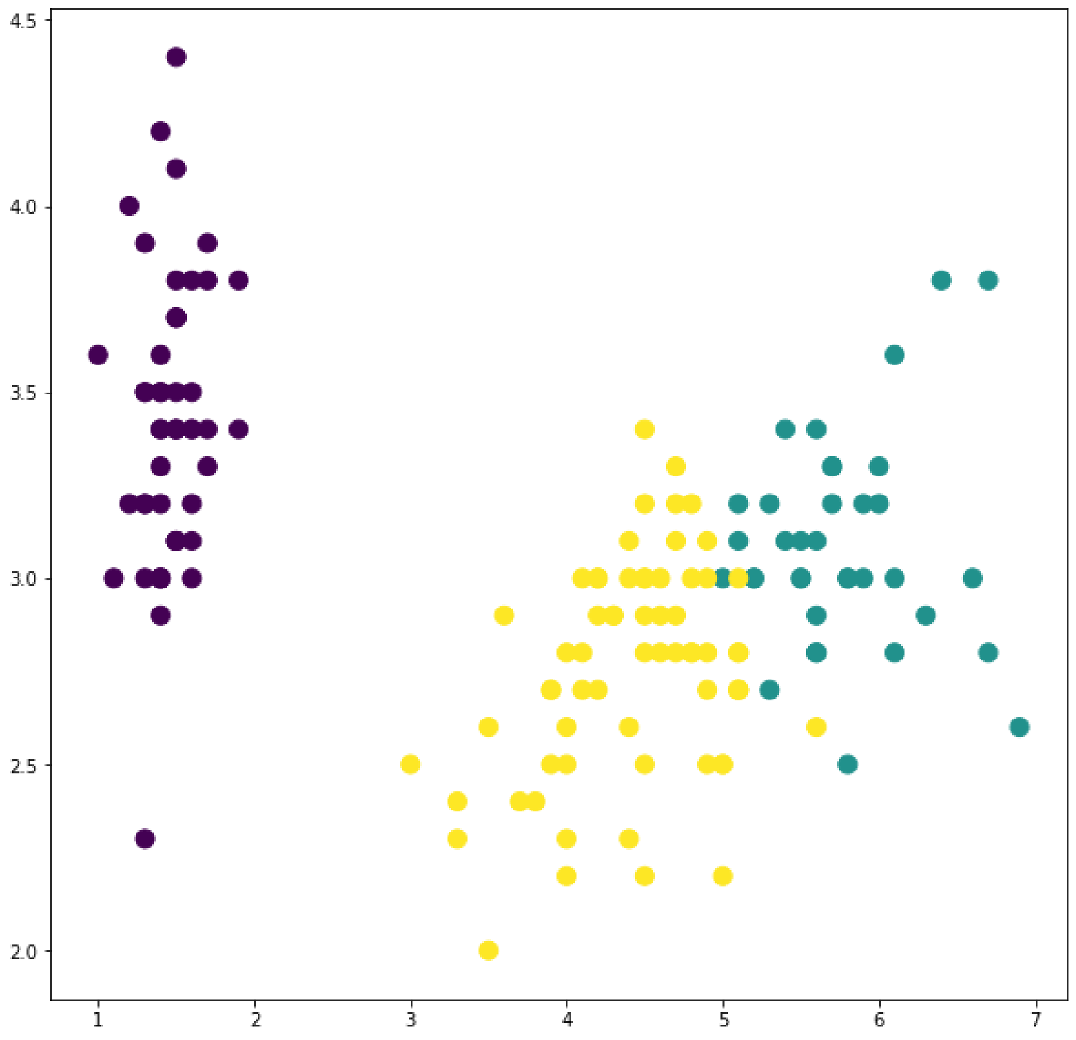
Now we apply the **Ward criterion** (uses the variances of the clusters)

```
[12]: %time clust = linkage(iris['data'], method='ward')
      plt.figure(figsize=(15,15))
      dendrogram(clust, distance_sort=True, orientation='right');
```

```
CPU times: user 1.87 ms, sys: 542 µs, total: 2.41 ms
Wall time: 2.01 ms
```



```
[13]: plt.figure(figsize=(10,10))
      clabels = fcluster(clust, 3, criterion='maxclust')
      plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=clabels,s=100);
```



This criteria scores a little lower than the previous one. As we do not usually have the labels with a real unsupervise dataset we will have to use other quality criteria to decide the method to use for clustering the data.

[14]:
```
print (adjusted_mutual_info_score(iris['target'], clabels))
```

```
0.7578034225092115
```

2.4 Conceptual Clustering

Related to hierarchical clustering but from the perspective of unsupervised machine learning, conceptual clustering ([49]) includes algorithms that build a hierarchy of concepts aimed to describe the groups of the data at different levels of abstraction.

These algorithms use as bias ideas from cognitive science about how people build their concept structures from their observation of the environment. There are different theories that try to explain how human categories form and are organized in the mind. One of these theories describes categorization as the construction of a hierarchy of concepts based on generality/specialization and described by probabilistic prototypes.

This theory explains some phenomenon observed in cognitive psychology experiments about human conceptualization. Mainly that we use a hierarchy to organize concepts, and we have a preferred level that we apply to categorize (*basic level*), and identify new examples and that concepts are not defined by necessary and sufficient conditions, but that we have prototypes that are used for this categorization. The decision about if an example belongs or not to a category is explained by the similarity to the prototype. This explains that there are examples that we consider more prototypical of a concept than others or that there are examples that are difficult to categorize because they share similarity to different prototypes.

The algorithms included in this area have some specific assumptions about how the learning of new concepts has to be. Learning is incremental in nature, not immediate, because experience is acquired from continuous observation. Also, concepts are learned jointly with their hierarchical relationships, and these are complex, so a hierarchy of concepts has to be polythetic (not binary).

These learning algorithms have to search in the space of hierarchies, so they begin with an empty hierarchy and at each step of the process a full hierarchy always exists. There is an objective function that measures the utility of the learned structure, that is used to decide how the hierarchy is modified when new examples appear. Finally, the updating of the structure is performed by a set of conceptual operators that decide what possible changes can be applied to the hierarchy.

Being the learning incremental, the final hierarchy depends on the order the examples are presented to the algorithm, this is plausible for modeling human learning, but it is not always practical from a data mining perspective.

The COBWEB algorithm ([45]) is an example of conceptual clustering algorithm. It is an incremental algorithm that builds a hierarchy of probabilistic concepts that act as prototypes (see figure 2.2). To build the hierarchy a heuristic measure called *category utility* (CU) is used. This is related to the *basic level* phenomenon described by cognitive psychology. This measure allows COBWEB to have as a first level of the hierarchy the concepts that better categorize a set of examples, the subsequent levels specialize this first level.

Defined for categorical attributes, category utility balances intra class similarity ($P(A_i = V_{ij}|C_k)$) and inter class similarity ($P(C_k|A_i = V_{ij})$). Combining both measures as trade off between the two measures for a given set of categories we have:

$$\sum_{k=1}^{K} P(A_i = V_{ij}) \sum_{i=1}^{I} \sum_{j=1}^{J} P(A_i = V_{ij}|C_k) P(C_k|A_i = V_{ij}) \tag{2.2}$$

Using Bayes theorem this measure can be transformed using only the information that can be estimated from the examples:

$$\sum_{k=1}^{K} P(C_k) \sum_{i=1}^{I} \sum_{j=1}^{J} P(A_i = V_{ij}|C_k)^2 \tag{2.3}$$

The term $\sum_{i=1}^{I} \sum_{j=1}^{J} P(A_i = V_{ij}|C_k)^2$ represents the number of attributes that can be correctly predicted for a category (cluster). We look for a partition that increases this number of attributes compared to a baseline (no partition):

$$\sum_{i=1}^{I} \sum_{j=1}^{J} P(A_i = V_{ij})^2 \tag{2.4}$$

The category utility measure compares the difference between having a specific partition of

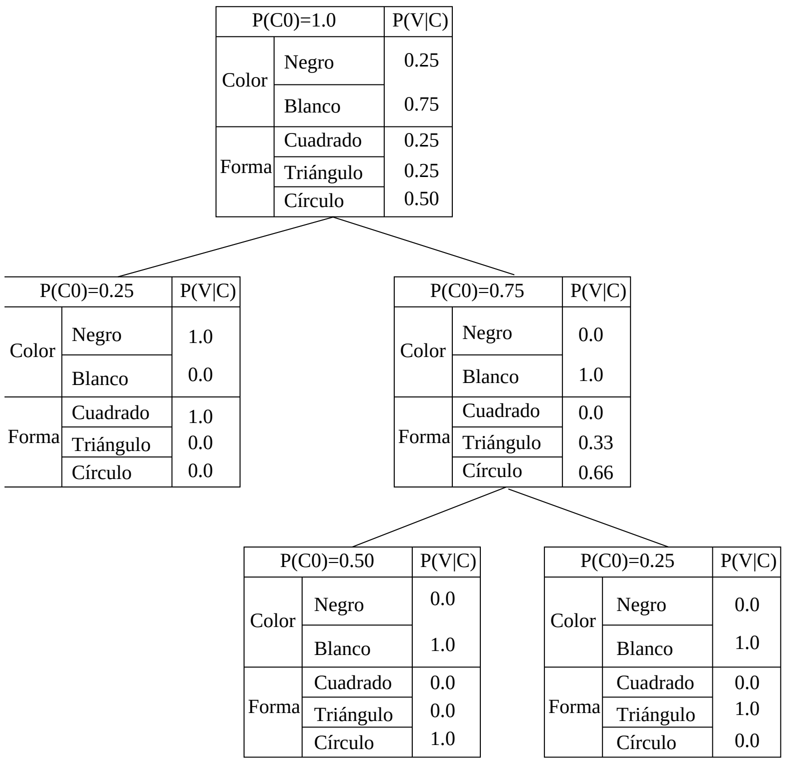


Figure 2.2: Example of COBWEB hierarchy of probabilistic concepts

the data and no having any partition at all in the following way:

$$CU(\{C_1,\ldots C_K\}) = \frac{\sum_{k=1}^{K} P(C_k) \sum_{i=1}^{I} \sum_{j=1}^{J} P(A_i = V_{ij}|C_k)^2 - \sum_{i=1}^{I} \sum_{j=1}^{J} P(A_i = V_{ij})^2}{K} \tag{2.5}$$

The measure scales by the number of partitions to be able to compare categorization of different sizes. This formula can be extended to continuous attributes assuming they have a specific distribution, for example for gaussian attributes:

$$CU(\{C_1,\ldots C_K\}) = \frac{\sum_{k=1}^{K} P(C_k) \sum_{i=1}^{I} \frac{1}{\sigma_i k} - \sum_{i=1}^{I} \frac{1}{\sigma_i p}}{K} \tag{2.6}$$

The algorithm (see algorithm 2.3) begins with an empty hierarchy and incrementally incorporates new examples traversing the hierarchy and modifying the concepts at each level using four conceptual operators.

- **Incorporate:** Put the example inside an existing class
- **New class:** Create a new class at this level
- **Merge:** Two concepts are merged, and the example is incorporated inside the new class (see figure 2.3)
- **Split:** A concept is substituted by its children (see figure 2.3)

The category utility is used to decide what conceptual operator to apply, by observing what modification of the concepts in a given level has the higher CU value. In order to reduce the

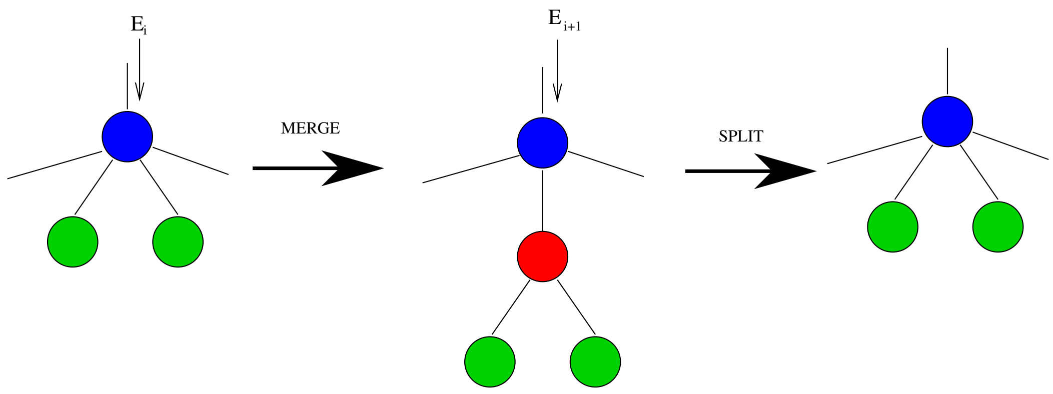


Figure 2.3: Merge and split operators of COBWEB

Algorithm 2.3 COBWEB algorithm

```
Procedure: Depth-first limited search COBWEB (x: Example, H: Hierarchy)
Update the father with the new example
if we are in a leaf then
    Create a new level with this example
else
    Compute CU of incorporating the example to each class
    Save the two best CU
    Compute CU of merging the best two classes
    Compute CU of splitting the best class
    Compute CU of creating a new class with the example
    Recursive call with the best choice
end
```

computational cost, not all the splits and merges are considered, only the two concepts with higher CU when including the new example can be merged and only the concept with higher CU can be split.

2.5 Partitional clustering Algorithms

Partitional clustering approximates the solution to finding the optimal partition of N objects in K groups. This problem is NP-hard, requiring the enumeration of all possible partitions of the objects for finding the optimal partition. There is a large variety of algorithms that perform this task using different criteria and information. The next subsections will review some of them including:

- Model/prototype based algorithms (K-means, Gaussian Mixture Models, Fuzzy K-means, Leader algorithm...)
- Density based algorithms (DBSCAN, DENCLUE...)
- Grid based algorithms (STING, CLIQUE...)
- Graph theory based algorithms (Spectral Clustering...)
- Other approaches: Affinity Clustering, Unsupervised Neural networks, SVM clustering

Algorithm 2.4 K-means algorithm

```
Algorithm: K-means (X: Examples, k:integer)
Generate k initial prototypes (e.g. k random examples)
repeat
    Reassign the examples to their nearest prototype
    Recalculate prototypes (centroids)
until no change in assignments or a number of iterations passes
```

2.5.1 Prototype/model based clustering

K-means and variants

Prototype and model based clustering assume that clusters fit to a specific shape, so the goal is to discover how different numbers of these shapes can explain the spatial distribution of the data.

The most used prototype based clustering algorithm is K-Means [39]. This algorithm assumes that clusters are defined by their center (the prototype), and they have spherical shapes. The fitting of these spheres is done by minimizing the distances from the examples to these centers. Examples are only assigned to one cluster, this is known as hard clustering or hard partitioning.

Different optimization criteria can be used to obtain the partition, but the most common one is the minimization of the sum of the square euclidean distance of the examples assigned to a cluster and the centroid of the cluster. The problem can be formalized as:

$$\min_{\mathcal{C}} \sum_{\mathcal{C}_i} \sum_{x_j \in \mathcal{C}_i} \| x_j - \mu_i \|_2^2 \tag{2.7}$$

This minimization problem is solved iteratively using a gradient descent algorithm, being k a parameter. Algorithm 2.4 outlines the canonical K-means algorithm. Its computational complexity depends on the number of examples (n), the number of features (f), the number of clusters (k), and the number of iterations that the algorithm performs until convergence (I), being linear on all these factors in time with a complexity $O(nfkI)$. The number of iterations can not be predicted apriori and depends on how cluster-like is the data. If the clusters are well separated and there are actually k clusters, the convergence is really fast. This algorithm needs all the data in memory, and the space needed for storing the clusters is negligible compared with the size of the dataset, so the space complexity is $O(nf)$. This spatial complexity makes this algorithm non-suitable for large datasets. Using secondary memory to store the data while executing the algorithm is not a good idea, because all the examples have to be check each iteration to see if their assignment to a cluster has changed.

Figure 2.4 presents a simple example of how the algorithm works. As can be seen, the first iteration assigns the examples to two randomly chosen centroids. As the clusters are well separated each iteration the new centroids are closer to the actual cluster centroids, converging in just tree steps of the algorithm in this case to the true clusters.

This algorithm has some drawbacks, being it an approximate algorithm, there is no guarantee about the quality of the solution. It converges to a local minimum that depends on the initial solution. Because of this, a common strategy to improve the quality of the solution is to run the algorithm from several random initializations keeping the best one. This increases the actual computational cost of the algorithm by a constant factor.

There are several alternatives to cope with this sensitivity to initialization. The original algorithm uses random chosen examples as initial centroids. The algorithm *K-means++* ([7]) proposes a seeding technique based on randomization that has proven to yield better results

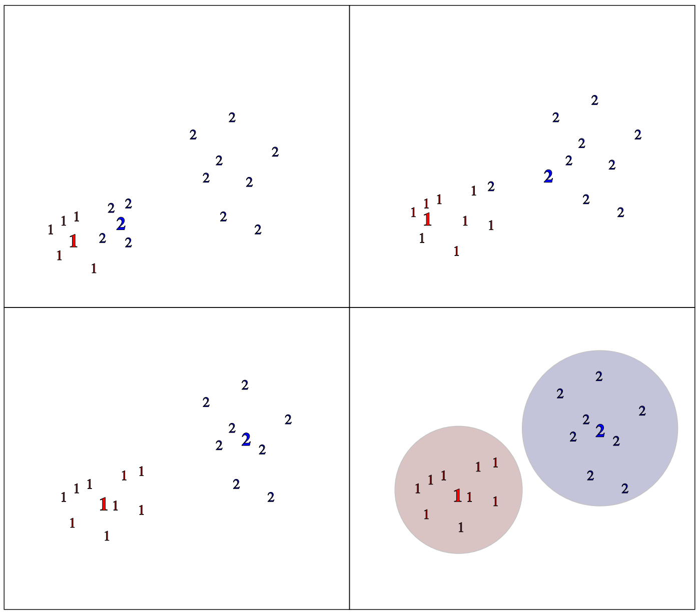


Figure 2.4: K-means algorithm example

and nowadays, it is the default option for most of the k-means implementation in widely used software libraries. The idea is to try to maximize the distance among initial centers, so they cover most of the space of examples. The algorithm is as follows:

1. Choose one centroid uniformly at random from among the data points
2. For each data point x, compute $d(x,c)$, the distance between x and the nearest centroid that has already been chosen
3. Choose one new data point at random as a new centroid, using a weighted probability distribution where a point x is chosen with probability proportional to $d(x,c)^2$
4. Repeat Steps 2 and 3 until k centroids have been chosen
5. Proceed with the standard K-means algorithm

K-means is also sensitive to clusters with different sizes and densities. Figure 2.5 shows an example of this problem. Both clusters are well separated, but one being small and compact and the other larger and spread makes that the best solution for K-means is to integrate part of the large cluster into the small one.

Outliers are also an issue for this algorithm, because it is possible that isolated examples would appear as one example clusters. This can be used as a feature of the algorithm as an alternative method for outlier detection, but having also in mind the problem of K-means with small clusters. If outliers are not really far from other densities, they will be integrated to other clusters, deviating the centroids from its true position.

Other practical problem of K-means, is how to determine k, the number of clusters. It is usually solved by exploring the range of different possibilities, but information from the domain can help to narrow it down. In the next chapter we will treat the problem of cluster validation that it is closely related to this problem ,and some criteria that help to determine how well a number of clusters fit the data will be discussed.

There are different variants of K-means that try to tackle some problems the original algorithm has or that include other methodologies that allow the discovery of other than spherical

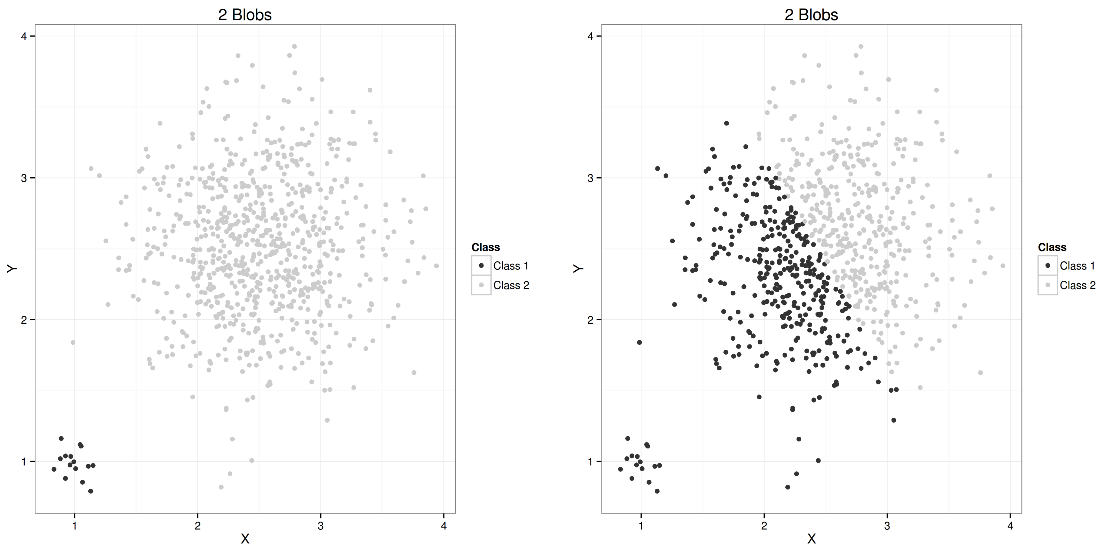


Figure 2.5: Clusters with different sizes and densities are problematic for K-means, left actual data labels, right partition obtained by k-means

clusters. The first one that is worth mentioning is *Bisecting K-means* ([120]). This algorithm begins with two clusters and iteratively decides which one of the current clusters to split, also in two, until the desired number of clusters is obtained. This makes that the clusters obtained are hierarchically nested being it similar to the hierarchical methods, but in this case there is not a complete hierarchy, only k splits are obtained. There are different criteria that can be used to decide what cluster is split in two. For instance, the size of the clusters can be used, so the largest cluster is the chosen one, or the variance of the distances of the examples to the centroids, so the cluster with larger spread is chosen. Other methods can be used, but the main idea is to use a criterion that selects the less adequate cluster from the current set, so the split results in better ones. The criteria could be also used to decide the parameter k, so the splitting can be performed until all the clusters have a similar quality, or the criteria has a value over a specific threshold. This is an outline of the algorithm:

1. Choose a number of partitions
2. Apply K-means to the dataset with k=2
3. Evaluate the quality of the current partition
4. Pick the cluster to split using a quality criteria
5. Apply K-means to the cluster with k=2
6. If the number of clusters is less than desired, repeat from step 3

Other interesting variant is *Global K-means* ([87]) that tries to minimize the clustering initialization dependence by exploring the clusterings that can be generated using all the examples as initialization points. For generating a partition with K clusters, it explores all the alternative partitions from 1 to K clusters, so it includes by construction the exploration of the range of possible number of clusters. It has been shown that this algorithm can generate better results than the original K-means but with a larger computational cost, because it has to run $K \times N$ times the K-means algorithm, so it is prohibitive for large datasets. There are some developments alleviating this temporal cost like [10] but with the cost of reducing the quality of the clusters. The following is an outline of the algorithm:

- Compute the centroid of the partition with 1 cluster
- For C from 2 to k:
  - for each example e, compute K-means initialized with the $C-1$ centroids from the previous iteration and an additional one with e as the C-th centroid
  - Keep the clustering with the best objective function as the C-clusters solution

Kernel K-means ([31]) is a kernelized version of the original algorithm where distances are computed in feature space using a kernel. This helps to discover non-spherical/non-convex clusters, transforming the original data space to one where the cluster shapes are more separable. The first problem is to determine what kernel is suitable for discovering the clusters, that leads to having to explore the different possibilities. Also, the computation of the kernel increases the computational cost, and the centroids that are used to cluster the data are in the feature space, so there is not an image in the original space that can be used to describe the cluster prototypes.

K-means assumes that the data is represented using continuous attributes, this means that a centroid can be computed, and it corresponds to a possible example. This is not always the case, sometimes data has nominal attributes, or it is structured (strings, graphs, sequences...), so a prototype computed as the mean of the examples of a cluster make no sense. In this case, the approach is to use one or more examples for each cluster as representative, this is called the medoid. Now the optimization criteria changes to optimizing the sum of distances from each example to the medoid of their cluster. This modifies the algorithm for updating the cluster representative, increasing the computational cost of the algorithm. In the case of only using one example as medoid, the cost per iteration for recomputing the medoid is $O(n^2)$, using more examples makes the algorithm NP-hard. This algorithm has the advantage of not being sensitive to outliers. The representative algorithm of this variation is *Partitioning Around Medoids* (PAM) ([79]), that works as follows:

1. Randomly select k of the n data points as the medoids
2. Associate each data point to the closest medoids
3. For each cluster m
   - For each non-medoid o in the cluster: Swap m's medoid and o and compute the cost
4. Keep the best solution
5. If medoids change, repeat from step 2

Leader algorithm

The *leader algorithm* is a simple clustering algorithm that has the advantage of being incremental and of a linear time complexity. This makes it suitable for processing streams of data and using different updating strategies to adapt the clusters to the changes in the model of the data, this is known as *concept drift*.

The parameter that controls this algorithm represents the radius of the sphere that encloses a cluster. This has an effect similar to the number of clusters in K-means, but in this case all the clusters are assumed to fit inside the same radius, meanwhile K-means does not constrain the size of the clusters.

Algorithm 2.5 Leader algorithm

```
Algorithm: Leader Algorithm (X: Examples, D:double)
Generate a prototype with the first example
while there are examples do
    e= current example
    d= distance of e to the the nearest prototype
    if $d \leq D$ then
        Introduce the example in the class
        Recompute the prototype
    else
        Create a new prototype with this example
    end
end
```

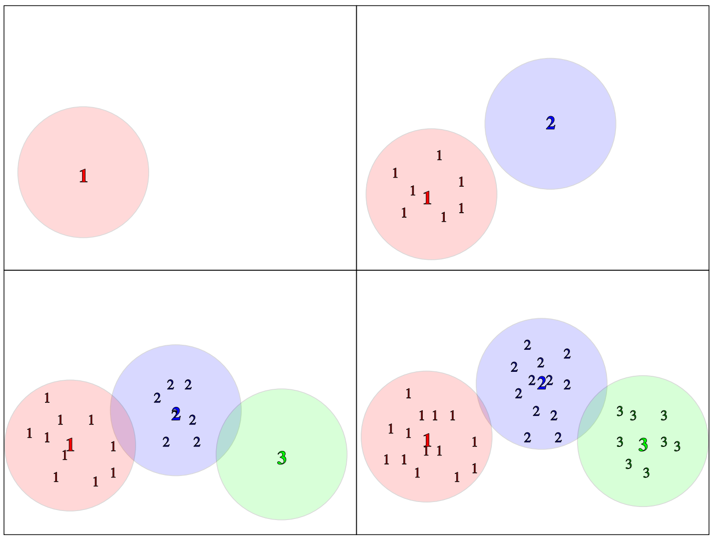


Figure 2.6: Leader algorithm example

Data is processed incrementally and clusters appear as needed, so the actual number of clusters emerges from the data. A new cluster is generated if there is no current cluster that includes the new example inside the radius. The new cluster will have as center this example. Usually the clusters are represented by the centroids, and there is no need for keeping all the seen examples if sufficient statistics are stored. Only storing the number of examples in the clusters, the sum of the examples and the squared sum of examples (that can be computed incrementally) allow for recomputing the new center. This makes this algorithm also space efficient. An outline of this algorithm appears in algorithm 2.5. Figure 2.6 shows an example of processing data by this algorithm, when new data is inside the radius of the clusters they are classified in an existing cluster, and new clusters appear when a new example that is not covered appears.

The quality of the clusters obtained by this algorithm is not usually as good as the one obtained by other algorithms, but it can be used as a preprocessing step for scalability, reducing the granularity of the data, so it can fit in memory, and then apply a better algorithm to the reduced data. The actual number of clusters that can be obtained using this algorithm is determined by the volume enclosed by the values of the attributes of the dataset. Because of the way this algorithm works, clusters may overlap, but the center of any cluster cannot be less

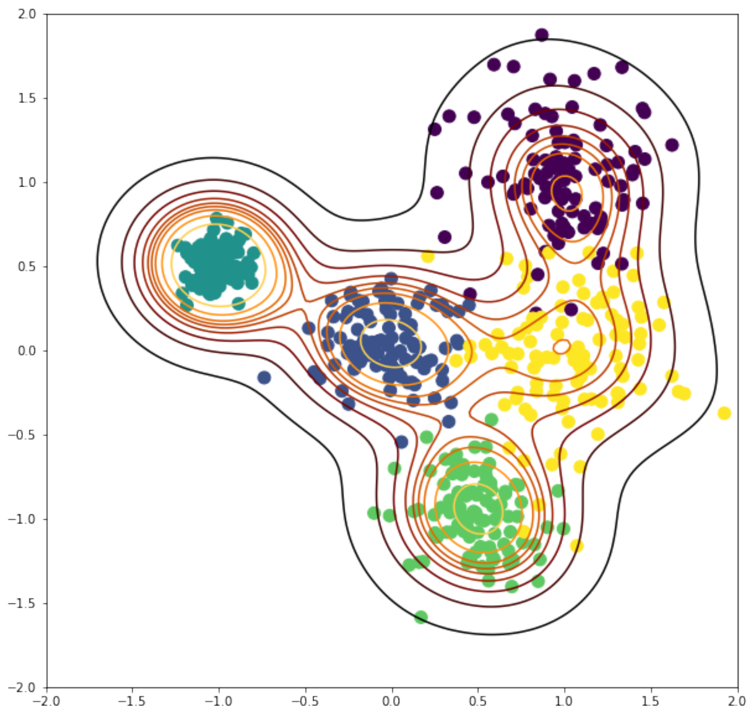


Figure 2.7: Clusters defined by bivariate gaussian distributions

than at a radius apart from any other cluster. This means that in any given dimension cannot be more than $(\frac{2 \cdot l}{r} - 1)$ centers in a given dimension being l the length of the dimension and r the clustering radius. This defines the granularity of the transformation and allows computing the maximum amount of memory that will be necessary to store the clustering.

Gaussian Mixtures

Model based clustering assumes that the dataset can be fit to a mixture of probability distributions. The shape of the clusters depends on the specific probability distribution used. A common choice is the gaussian distribution. In this case, depending on the choice about if covariance among attributes are modeled or not, the clusters correspond to arbitrarily oriented ellipsoids or spherical shapes (see figure 2.7). The model fit to the data can be expressed in general as:

$$P(x|\theta) = \sum_{k=1}^{K} w_k P(x|\theta_k) \tag{2.8}$$

With K the number of clusters, and $\sum_{k=1}^{K} w_k = 1$. This model is usually fit using the Expectation-Maximization algorithm (EM), assigning to each example a probability to each cluster, meaning that we are obtaining a soft clustering of the data.

The main limitation of this method is to assume that the number of partition is known. Also, it needs to have all the data in memory for performing the computations, so their spatial needs scale linearly with the size of the dataset. The storage of the model is just a fraction of the size of the dataset for prototype based algorithms, but for model based algorithms, it depends on the number of parameters needed to estimate the probability distributions. This number grows quadratically with the number of attributes if, for example, gaussian distributions with full covariance matrices are used as model.

As we discussed in the section about K-means, the computational time cost for the prototype based algorithms each iteration is $O(nfk)$. For the model based algorithms, each iteration has to estimate all the parameters (p) of all the distributions of the model (k) for all instances (n), the number of parameters is proportional to the number of features and also depends on what assumptions make about the independence of the attributes. In the case of full covariance estimation, each iteration results in a total time complexity of $O(nf^2k)$, if we assume attribute independence time complexity is $O(nfk)$.

For learning this model the goal is to estimate the parameters of the distribution that describes each class (e.g.: means and standard deviations). The Expectation Maximization (EM) algorithm is the usual choice. It maximizes the likelihood of the distribution respect the dataset iteratively, beginning with an initial estimate of the parameters. It repeatedly performs two steps until convergence (no further improvements of the likelihood):

- **Expectation:** a function that assigns a degree of membership to all the instances to any of the K probability distributions is computed using the current estimation of the parameters of the model.
- **Maximization:** the parameters of the distributions are reestimated using as weights the memberships obtained from the previous step to maximize the memberships of the examples to the new model.

Each specific model has a set of parameters that have to be estimated using the EM algorithm. In the case of choosing the Gaussian distribution, parameters are the mean and covariance matrix, so the model is:

$$P(x|\overrightarrow{\mu},\Sigma) = \sum_{k=1}^{K} w_k P(x|\overrightarrow{\mu_k},\Sigma_k) \tag{2.9}$$

The actual computations depend on the assumptions that we make about the attributes (independent or not, same $\sigma\ldots$). If the attributes are assumed independent μ_i and σ_i have to be computed for each class ($O(k)$ parameters), the model is described by hyper spheres if we also constrain the model so all the attributes of each cluster have the same variance or ellipsoids parallel to coordinate axis if each attribute has its own variance. If the attributes are modeled as not independent μ_i, σ_i and σ_{ij} have to be computed for each class ($O(k^2)$ parameters). The model is described by hyper ellipsoids non-parallel to coordinate axis, and we can choose between a common covariance matrix, so all hyper ellipsoids are identical, or to have a covariance matrix for each cluster.

For the case of A independent attributes, each cluster has the model:

$$P(x|\overrightarrow{\mu_k},\Sigma_k) = \prod_{j=1}^{A} P(x|\mu_{ij},\sigma_{ij}) \tag{2.10}$$

With the model to fit:

$$P(x|\overrightarrow{\mu},\overrightarrow{\sigma}) = \sum_{k=1}^{K} w_k \prod_{j=1}^{A} P(x|\mu_{kj},\sigma_{kj}) \tag{2.11}$$

The update rules used by the EM algorithm for the model parameters are obtained from the log likelihood of the model, by differentiating this function for each parameter and finding where the gradient is zero. For the case of gaussian distributions the expectation step computes the weights corresponding for each gaussian to each example that represent the probability that an example x_i is generated by component C_k:

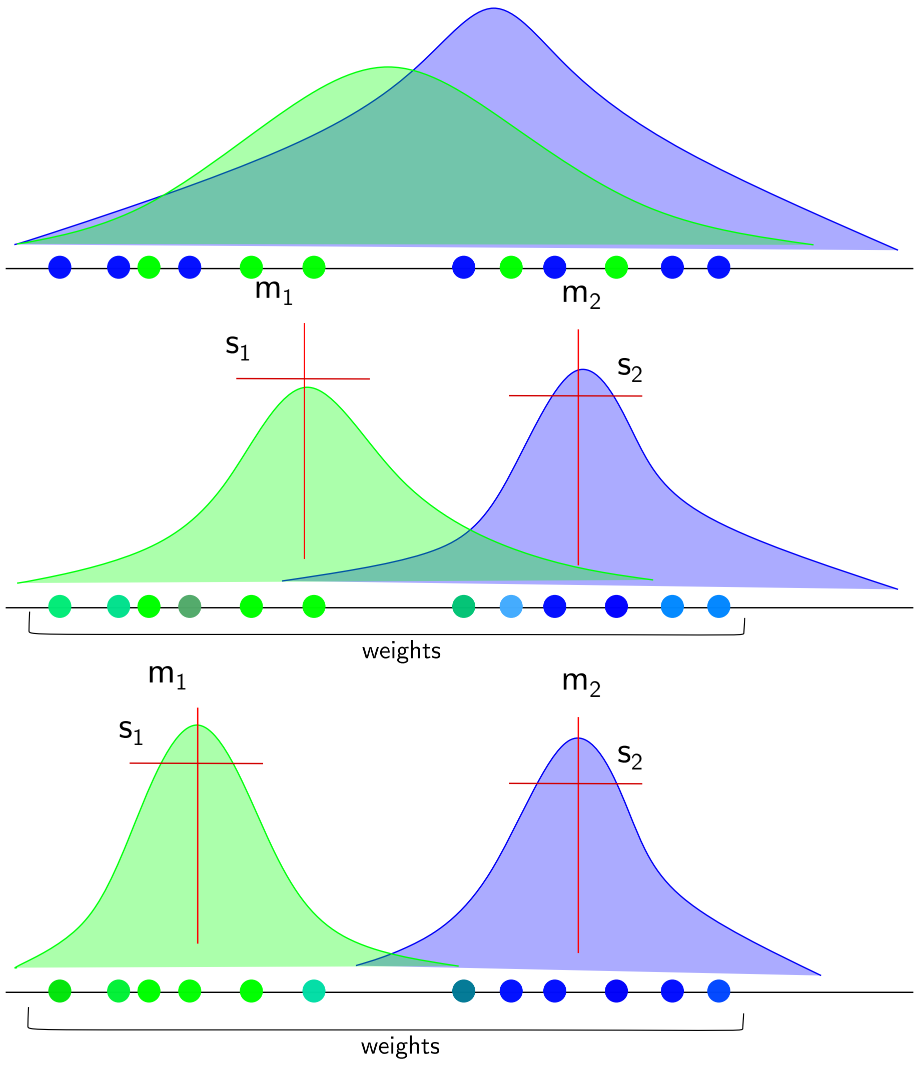


Figure 2.8: Example of the EM algorithm

$$\hat{\gamma}_{ik} = w_k P(x_i|\mu_k, \sigma_k) \tag{2.12}$$

the update equations of the parameters in the maximization step are:

$$\hat{\mu}_k = \frac{\sum_{i=1}^{N} \hat{\gamma}_{ik} x_i}{\sum_{i=1}^{N} \hat{\gamma}_{ik}} \tag{2.13}$$

$$\hat{\sigma}_k = \frac{\sum_{i=1}^{N} \hat{\gamma}_{ik} (x_k - \hat{\mu}_i)^2}{\sum_{i=1}^{N} \hat{\gamma}_{ik}} \tag{2.14}$$

$$\hat{w}_k = \frac{1}{N} \sum_{k=1}^{N} \hat{\gamma}_{ik} \tag{2.15}$$

The EM algorithm is initialized with a set of K initial distributions usually obtained with K-means. The μ_i and $\sigma_{i,j}$ for each cluster and attribute are estimated from the hard partition. The algorithm iterates until there is not improvement in the log likelihood of the model given the data.

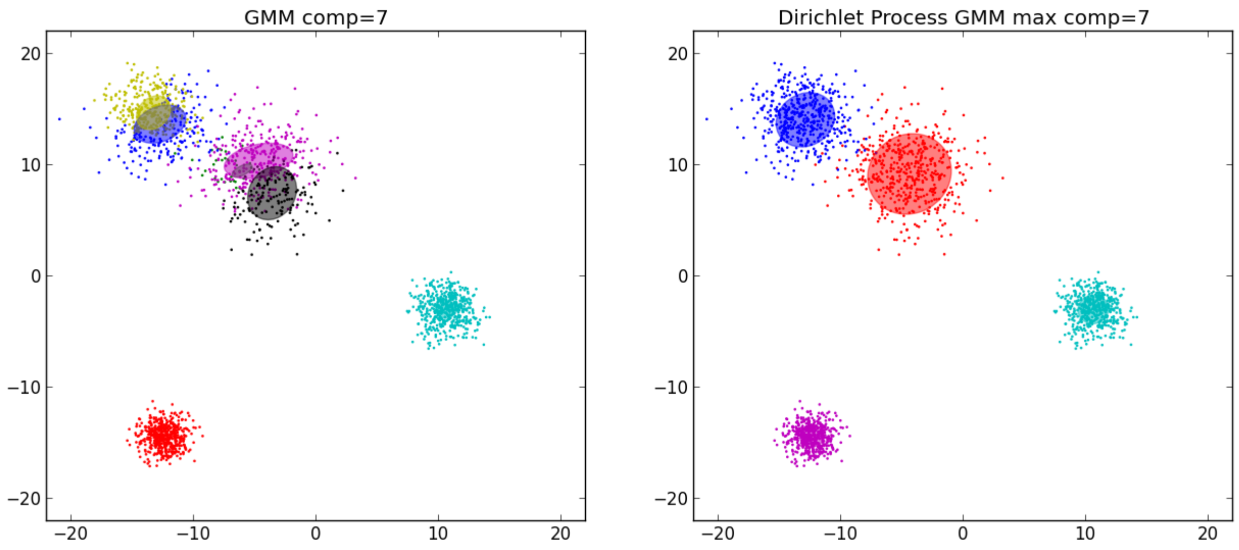


Figure 2.9: plain GMM versus Dirichelet Process GMM

Figure 2.8 shows three snapshots of the algorithm for a set of univariate data. The first plot shows the initialization, where two arbitrary gaussians are used as the initial clusters. After computing the probabilities, for each example and each distribution the parameters are reestimated, changing the distributions. The colors in the second plot represent the membership for each example to each gaussian and now the distributions are closer to the data. The third plot represents the final step when the reestimation does not change the parameters of the distribution anymore. Now the colors are more similar because the variance of the distributions is smaller, but the examples at the border of the clusters still have some probability of belonging to the othe cluster.

It can be shown that K-means is a particular case of this algorithm where only the means of the distributions are estimated and each example has only weights $1/0$ determined by the closest centroid. The main advantage of GMM is that we obtain a membership as a probability (soft assignments) that can be useful for some tasks. Also, using different probability distributions we can find other kinds of structures in the data, not only spherical ones. The main drawback is the derivation of the update rules for the EM algorithm. In the case of gaussian distributions there is a closed form for the updates as we saw previously, but other models have to resort to other methods for the estimation, like Monte Carlo Markov Chains (MCMC) or variational inference, increasing the computational cost of the maximization step.

Just as in K-means, the main issue of GMM is to decide a priori the number of components. The mixture model paradigm gives more flexibility and allows introducing the number of clusters as a part of the estimation parameters. This is introduced by adding to the model a prior over the number of components of the mixture described as a Dirichlet Process distribution. This distribution makes the assumption of an unbound number of components, introducing a finite weight that is distributed among all the components. The estimation process allocates the examples and distributes the weights to the components deciding what number of components better suits the data. This method is known as the Dirichlet Process Gaussian Mixture Model. Figure 2.9 shows a GMM, and a Dirichlet Process GMM fitted to a dataset with only 4 clusters but using 7 as the guess of the number of clusters. As can be seen the Dirichlet process GMM nullifies the weights of the extra components while the GMM divides some cluster to obtain the requested number of partitions.

Fuzzy C-means

Fuzzy clustering relax the hard partition constraint of K-means like gaussian mixture models, but in this case fuzzy membership functions are used for distributing the assignment of examples to clusters, so we have soft clusters in this case. This is an advantage from other algorithms when the groups are overlapped.

The Fuzzy clustering algorithm optimizes the following function:

$$L = \sum_{i=1}^{N} \sum_{k=1}^{K} \delta(C_k, x_i)^b \|x_i - \mu_k\|^2 \tag{2.16}$$

where $\sum_{k=1}^{K} \delta(C_k, x_i) = 1$ and b is a blending factor that allows a continuum between hard and soft partitions. Notice that the summation is for all the examples for each cluster instead of summing up for only the examples assigned to a cluster as in k-means. This is similar to what is done in gaussian mixture models, but in this case we use a membership function that is proportional to the distances to the centroids.

Fuzzy C-means is the most known fuzzy clustering algorithm, and corresponds to a fuzzy version of K-means. The algorithm performs an iterative optimization of the above objective function, updating of the cluster centers using the following equation:

$$\mu_j = \frac{\sum_{i=1}^{N} \delta(C_j, x_i)^b x_i}{\sum_{i=1}^{N} \delta(C_j, x_i)^b} \tag{2.17}$$

and the updating of the memberships of the examples to the clusters as:

$$\delta(C_j, x_i) = \frac{(1/d_{ij})^{1/(1-b)}}{\sum_{k=1}^{K} (1/d_{ik})^{1/(1-b)}}, \; d_{ij} = \|x_i - \mu_j\|^2 \tag{2.18}$$

This behavior can be assimilated to the EM algorithm used by GMM.

The bias of the C-means algorithm is to look for spherical clusters like the previous algorithms, but there are other alternative optimization functions that allow for more flexible shapes, but increasing the computational cost, like for instance:

- Gustafson-Kessel algorithm: A covariance matrix is introduced for each cluster in the objective function that allows elipsoidal shapes and different cluster sizes
- Gath-Geva algorithm: Adds to the objective function the size, and an estimation of the density of the cluster

This method allows also for using other objective functions that look for specific geometrical shapes in the data (lines, rectangles...) making this characteristic specially useful for image segmentation and recognition.

2.5.2 Notebook: Prototype/Model Based Clustering

We are going to apply some of the model based clustering algorithms to the iris dataset and we will show also some of the shortcomings of the K-means algorithm

[2]:
```python
from sklearn import datasets
from sklearn.metrics import adjusted_mutual_info_score
from scipy.cluster.hierarchy import linkage, dendrogram, fcluster
```

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from kemlglearn.cluster import Leader, KMedoidsFlexible
from kemlglearn.datasets import make_blobs
from numpy.random import normal
import numpy as np
```

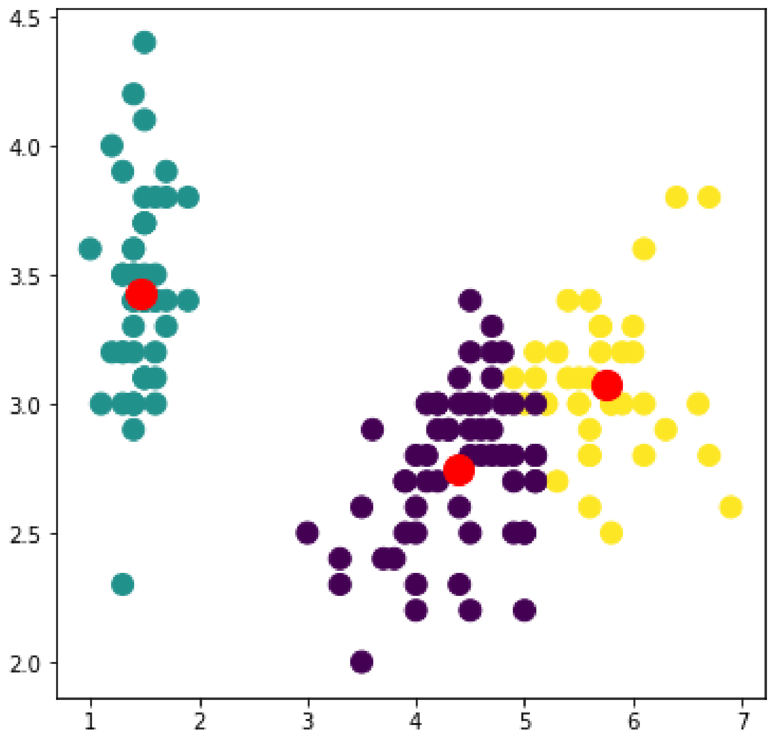
K-means

[3]:
```
iris = datasets.load_iris()
```

We will play again with the iris dataset, now using K-means. In this case we will look for 3 clusters, as we actually know the number of classes.

[4]:
```
km = KMeans(n_clusters=3)
%time {km.fit(iris['data'])}
labels = km.predict(iris['data'])
plt.figure(figsize=(6,6))
plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=labels, s=100)
plt.scatter(km.cluster_centers_[:,2], km.cluster_centers_[:,1], c='r',␣
  ↪s=200);
```

```
CPU times: user 82.2 ms, sys: 782 µs, total: 83 ms
Wall time: 83.3 ms
```



Results are a little bit worse than hierarchical clustering.

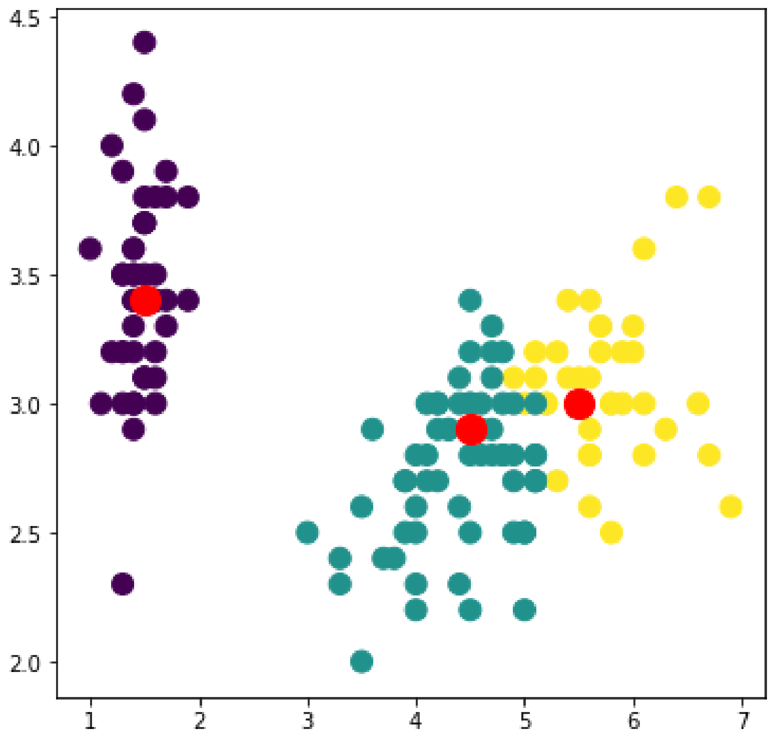
[5]:
```
print(adjusted_mutual_info_score(iris['target'], labels))
```

```
0.7483723933229485
```

K-medoids

```
[6]: kmd = KMedoidsFlexible(n_clusters=3)
     %time {kmd.fit(iris['data'])}
     labels = kmd.predict(iris['data'])
     plt.figure(figsize=(6,6))
     plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=labels, s=100)
     plt.scatter(kmd.cluster_medoids_[:,2], kmd.cluster_medoids_[:,1], c='r',␣
      ↪s=200);
```

```
CPU times: user 2.5 s, sys: 38.5 ms, total: 2.54 s
Wall time: 2.43 s
```



```
[7]: print(adjusted_mutual_info_score(iris['target'], labels))
```

```
0.7483723933229484
```

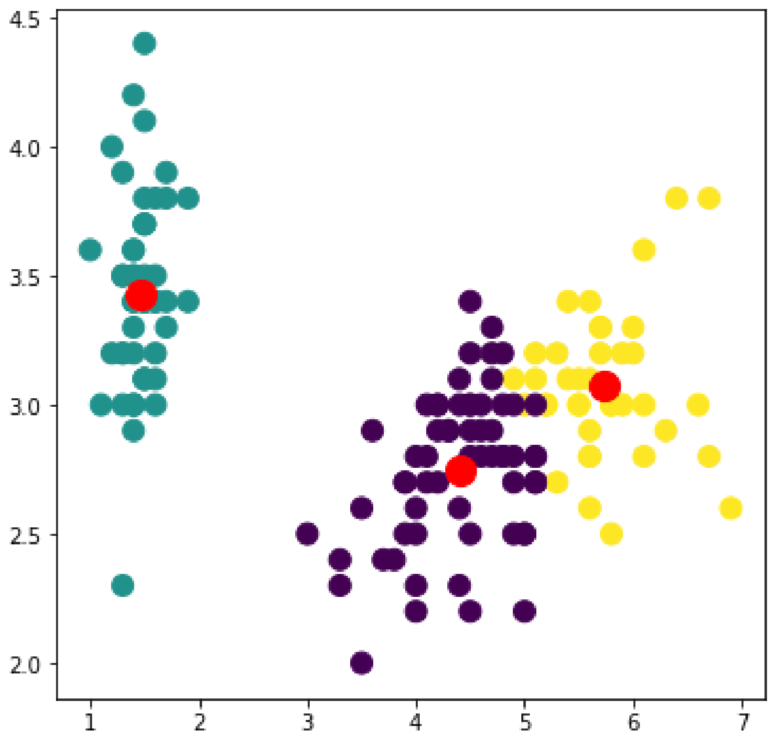
GMM

Now for Gaussian Mixture Models, first using spherical clusters as estimation method.

```
[8]: gmm = GaussianMixture(n_components=3, covariance_type='spherical')
     %time {gmm.fit(iris['data'])}
     labels = gmm.predict(iris['data'])
```

```
plt.figure(figsize=(6,6))
plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=labels, s=100);
plt.scatter(gmm.means_[:,2], gmm.means_[:,1], c='r', s=200);
```

```
CPU times: user 4.2 ms, sys: 929 µs, total: 5.13 ms
Wall time: 4.81 ms
```



These are the results of the AMI and the final BIC of the model

```
[9]: print("AMI=", adjusted_mutual_info_score(iris['target'], labels))
     print("BIC=", gmm.bic(iris['data']))
```

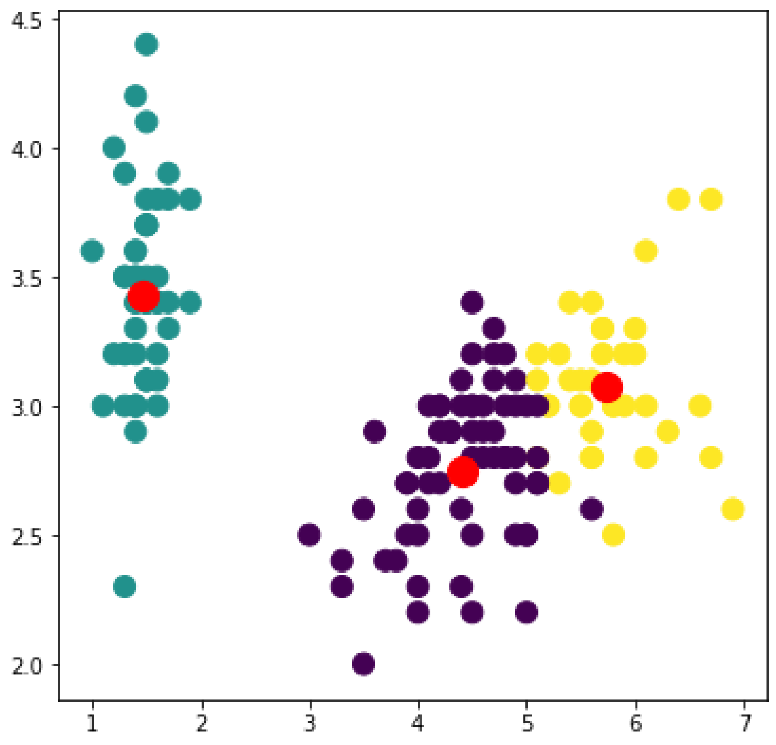
```
AMI= 0.7483723933229485
BIC= 853.809340502941
```

As expected the results is comparable to the one from K-means.

Let's change the method of estimation assuming independent attributes (diagonal covariance).

```
[10]: gmm = GaussianMixture(n_components=3, covariance_type='diag')
      %time {gmm.fit(iris['data'])}
      labels = gmm.predict(iris['data'])
      plt.figure(figsize=(6,6))
      plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=labels, s=100)
      plt.scatter(gmm.means_[:,2], gmm.means_[:,1], c='r', s=200);
```

```
CPU times: user 12.9 ms, sys: 1.17 ms, total: 14.1 ms
Wall time: 11.9 ms
```



These are the results of the AMI and the final BIC of the model

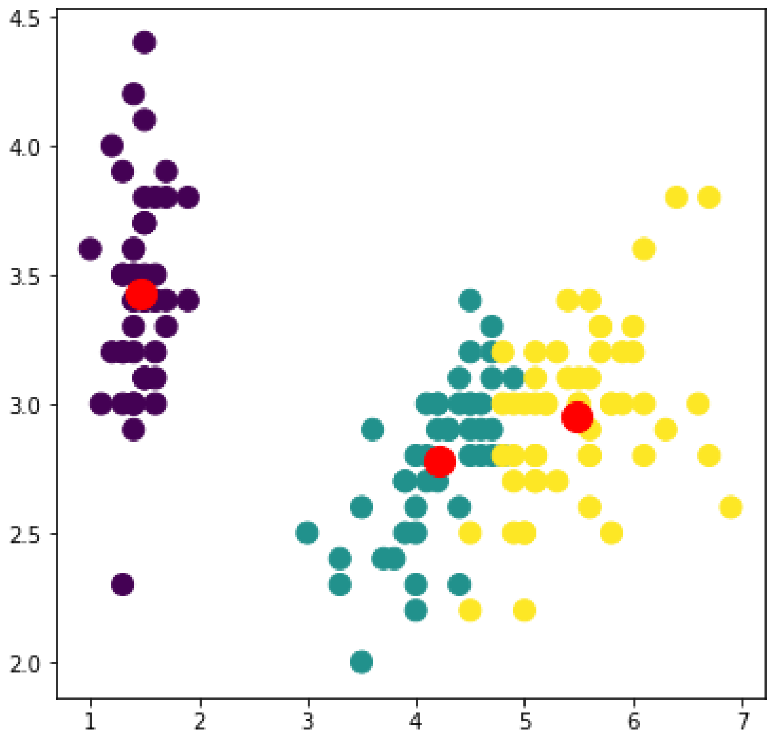
```
[11]: print("AMI=", adjusted_mutual_info_score(iris['target'], labels))
      print("BIC=", gmm.bic(iris['data']))
```

```
AMI= 0.7934250515435665
BIC= 744.6332089584272
```

Now we change to the full model, with dependent attributes

```
[12]: gmm = GaussianMixture(n_components=3, covariance_type='full')
      %time {gmm.fit(iris['data'])}
      labels = gmm.predict(iris['data'])
      plt.figure(figsize=(6,6))
      plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=labels, s=100)
      plt.scatter(gmm.means_[:,2], gmm.means_[:,1], c='r', s=200);
```

```
CPU times: user 27.3 ms, sys: 156 µs, total: 27.4 ms
Wall time: 25.4 ms
```



These are the results of the AMI and the final BIC of the model

```
[13]: print("AMI=", adjusted_mutual_info_score(iris['target'], labels))
      print("BIC=", gmm.bic(iris['data']))
```

```
AMI= 0.8970537476260634
BIC= 580.8594247694391
```

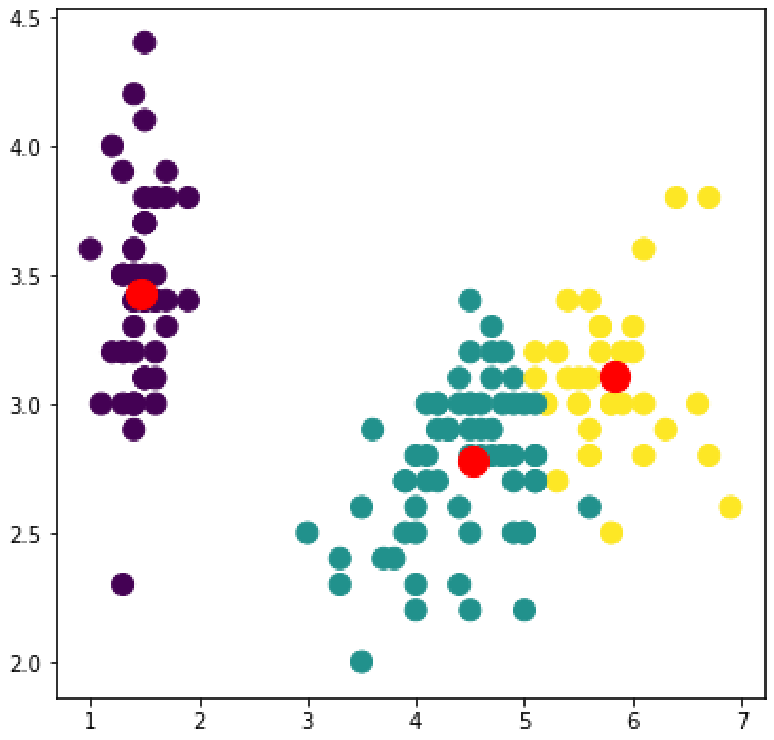
This is now the **best model** we have found

Leader Algorithm

Now we use the Leader Algorithm (from kemlglearn), the main problem is to guess a radius that results in the number of clusters we want and the quality could not be the best, the upside is that this algorithm is faster than the rest.

```
[14]: r=2.5
      lead = Leader(radius=r)
      %time {lead.fit(iris['data'])}
      labels = lead.predict(iris['data'])
      print("AMI=", adjusted_mutual_info_score(iris['target'], labels))
      plt.figure(figsize=(6,6))
      plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=labels, s=100)
      plt.scatter(lead.cluster_centers_[:,2], lead.cluster_centers_[:,1], c='r',␣
       ↪s=200);
```

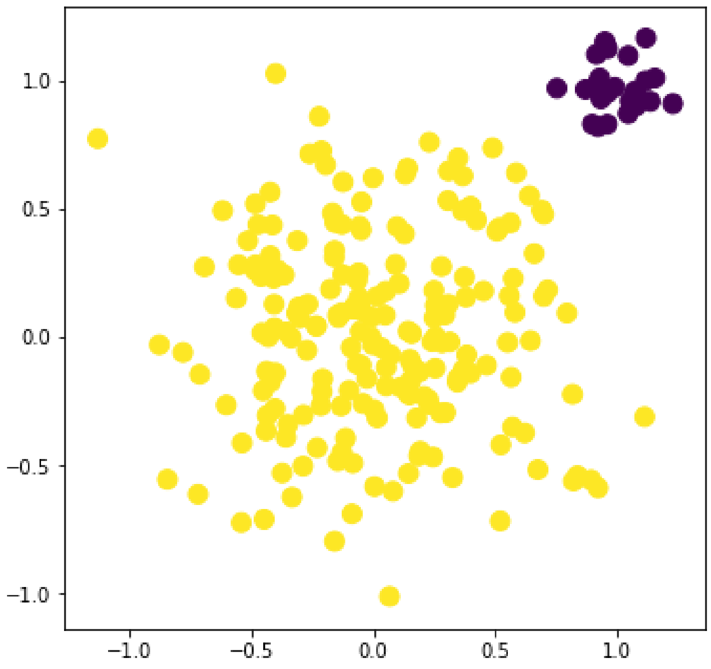
```
CPU times: user 51 ms, sys: 1.79 ms, total: 52.8 ms
Wall time: 47.4 ms
AMI= 0.7842528489695738
```



Issues with K-means

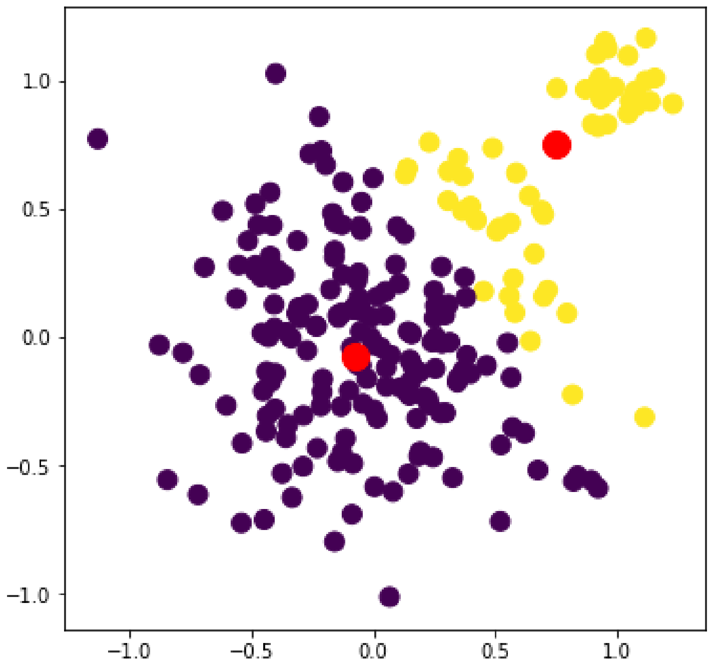
Now we will generate artificial data to see some issues of K-means. The first problem appears when the clusters have different sizes and variances. Depending on the ratio of sizes and variance difference, part of the examples from the larger cluster could be assigned to the small one.

[15]:
```
sc1=25
vc1=0.1
sc2=200
vc2=0.4

blobs, blabels = make_blobs(n_samples=[sc1,sc2], n_features=2,␣
 ↪centers=[[1,1], [0,0]], cluster_std=[vc1,vc2])
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111)
plt.scatter(blobs[:, 0], blobs[:, 1], c=blabels, s=100);
```



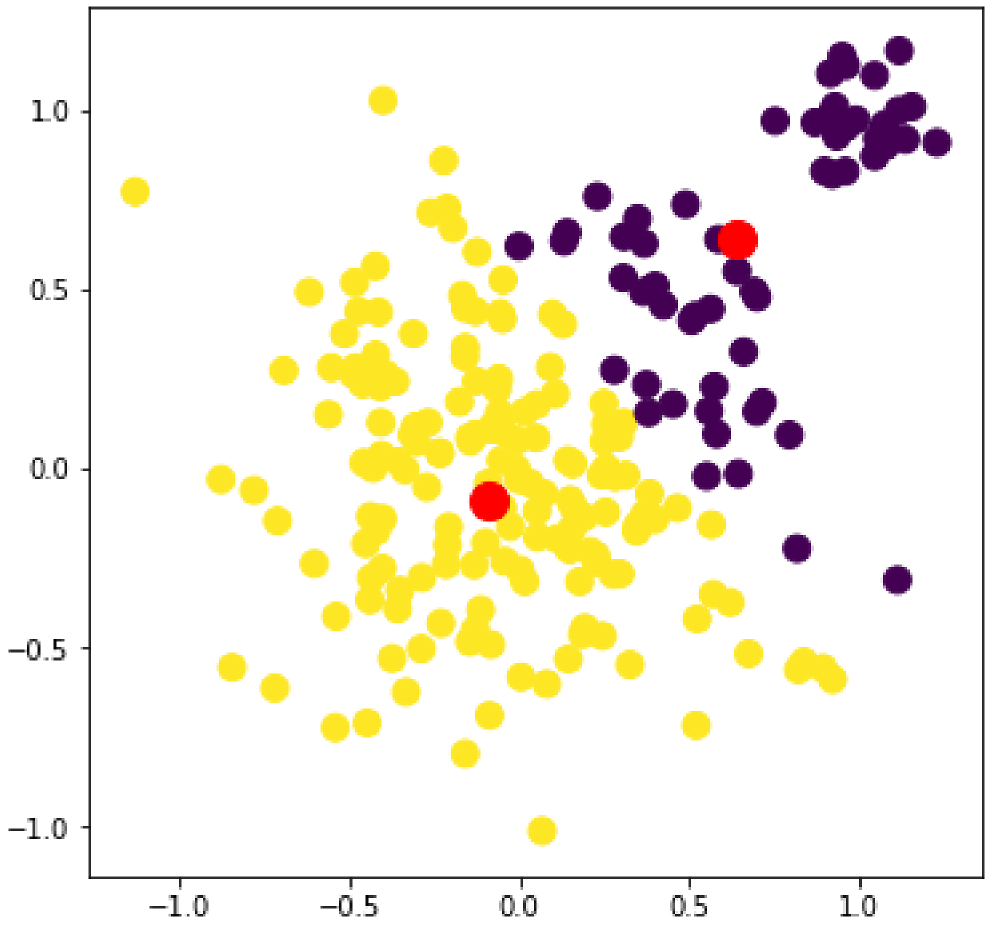
```
[16]: km = KMeans(n_clusters=2)
      km.fit(blobs)
      labels = km.fit_predict(blobs)
      print("AMI=", adjusted_mutual_info_score(blabels, labels))
      plt.figure(figsize=(6,6))
      plt.scatter(blobs[:, 0], blobs[:, 1], c=labels, s=100)
      plt.scatter(km.cluster_centers_[:,0], km.cluster_centers_[:,0], c='r',␣
       ↪s=200);
```

```
AMI= 0.3295229366550898
```



[17]:
```
kmd = KMedoidsFlexible(n_clusters=2, max_iter=100)
kmd.fit(blobs)
labels = kmd.fit_predict(blobs)
print("AMI=", adjusted_mutual_info_score(blabels, labels))
plt.figure(figsize=(6,6))
plt.scatter(blobs[:, 0], blobs[:, 1], c=labels, s=100)
plt.scatter(kmd.cluster_medoids_[:,0], kmd.cluster_medoids_[:,0], c='r',␣
 ↪s=200);
```

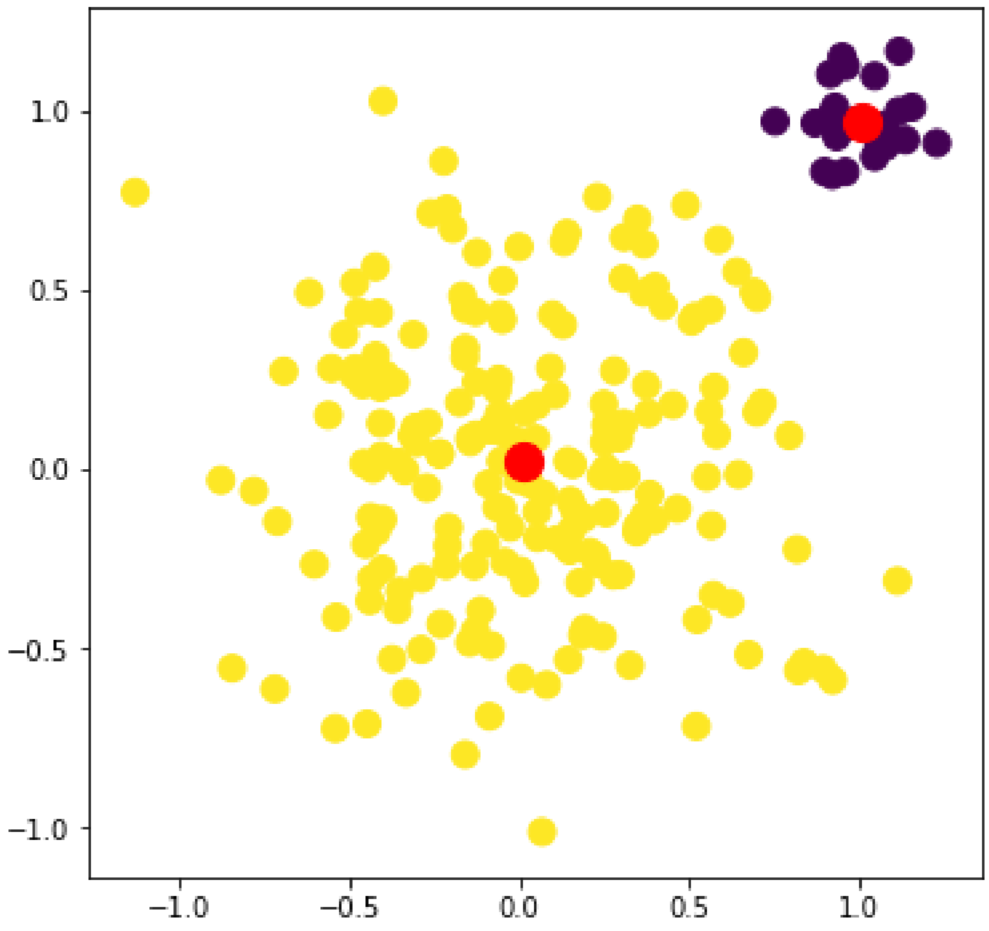
```
AMI= 0.2928814577591342
```



This could affect less to the GMM algorithm, but it is not free of this problem.

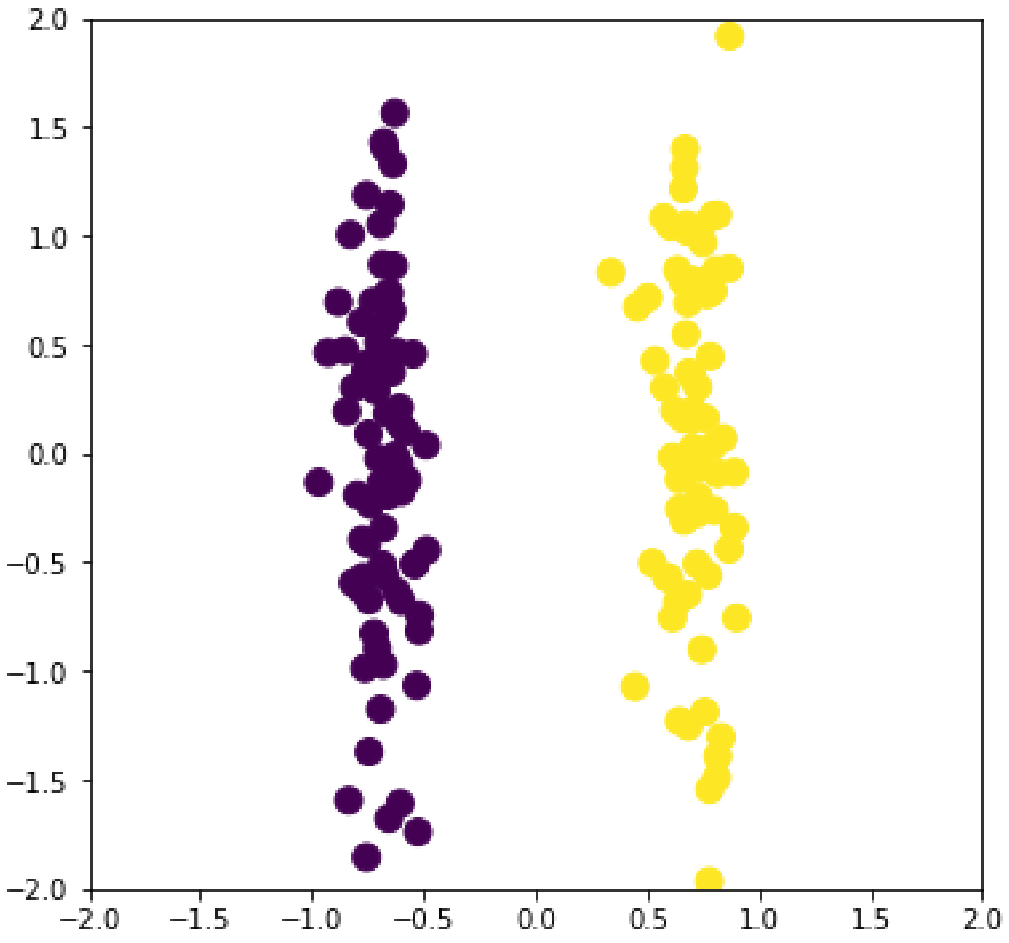
[18]:
```
gmm = GaussianMixture(n_components=2, covariance_type='diag')
gmm.fit(blobs)
labels = gmm.predict(blobs)
print("AMI=",adjusted_mutual_info_score(blabels, labels))
plt.figure(figsize=(6,6))
plt.scatter(blobs[:, 0], blobs[:, 1], c=labels, s=100)
plt.scatter(gmm.means_[:,0], gmm.means_[:,1], c='r', s=200);
```

```
AMI= 1.0
```



Problems also appear when the clusters are not spherical and some attributes have more variance than others (actually are elipsoids). If the clusters are not well separated, the partition can result in not very natural clusters. You can change this data set and see what happens if you change the size, separation and variance of the clusters.

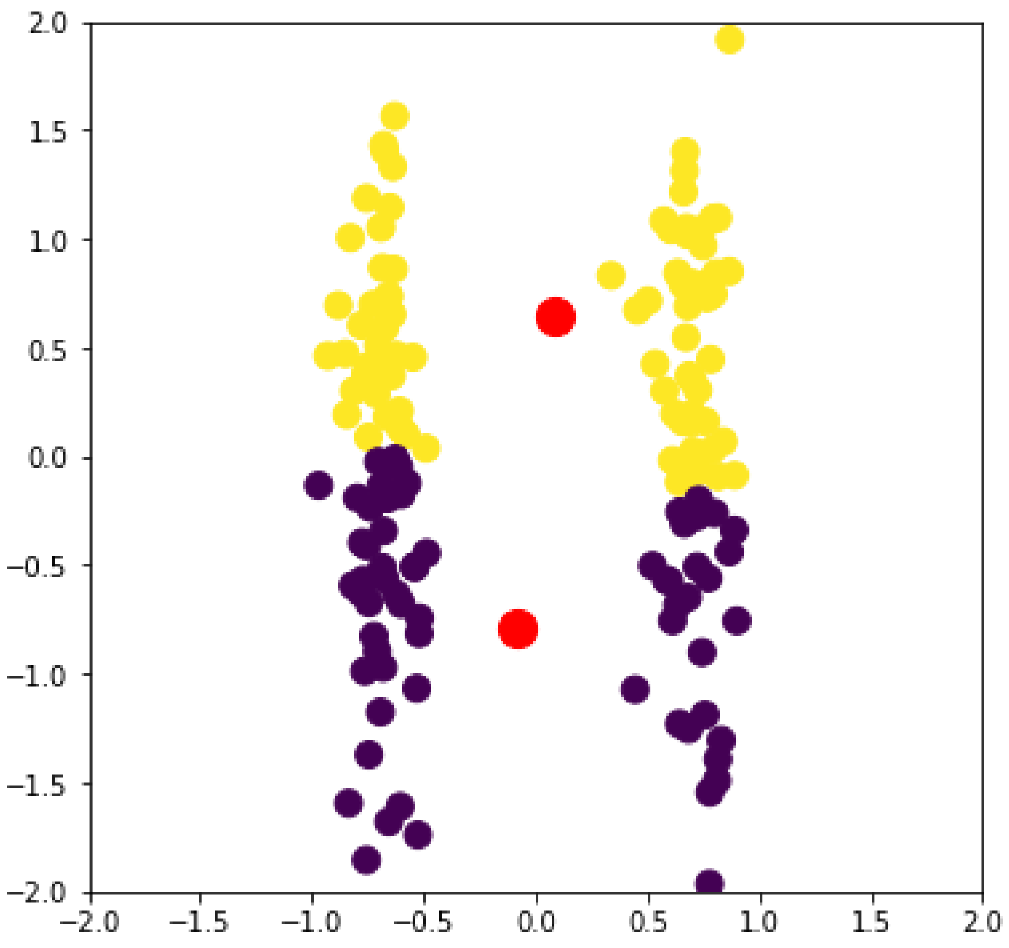
[19]:
```
sc1=75
v1=0.1
sc2=75
v2=0.9

data = np.zeros((sc1+sc2,2))
data[0:sc1, 0] = normal(loc=-0.7, scale=v1, size=sc1)
data[0:sc1, 1] = normal(loc=0.0, scale=v2, size=sc1)
data[sc1:, 0] = normal(loc=0.7, scale=v1, size=sc2)
data[sc1:, 1] = normal(loc=0.0, scale=v2, size=sc2)
dlabels = np.zeros(sc1+sc2)
dlabels[sc1:] = 1
plt.figure(figsize=(6,6))
plt.ylim(-2,2)
plt.xlim(-2,2)
plt.scatter(data[:, 0], data[:, 1], c=dlabels, s=100);
```



K-means divides the two clusters wrong (sometimes depends on the initialization, you can use the random_state parameter of K-means to see if different initializations get it right)

[20]:
```
km = KMeans(n_clusters=2)
labels = km.fit_predict(data)
print("AMI=", adjusted_mutual_info_score(dlabels, labels))
plt.figure(figsize=(6,6))
plt.ylim(-2,2)
plt.xlim(-2,2)
plt.scatter(data[:, 0], data[:, 1], c=labels, s=100)
plt.scatter(km.cluster_centers_[:,0], km.cluster_centers_[:,1], c='r',␣
 ↪s=200);
```

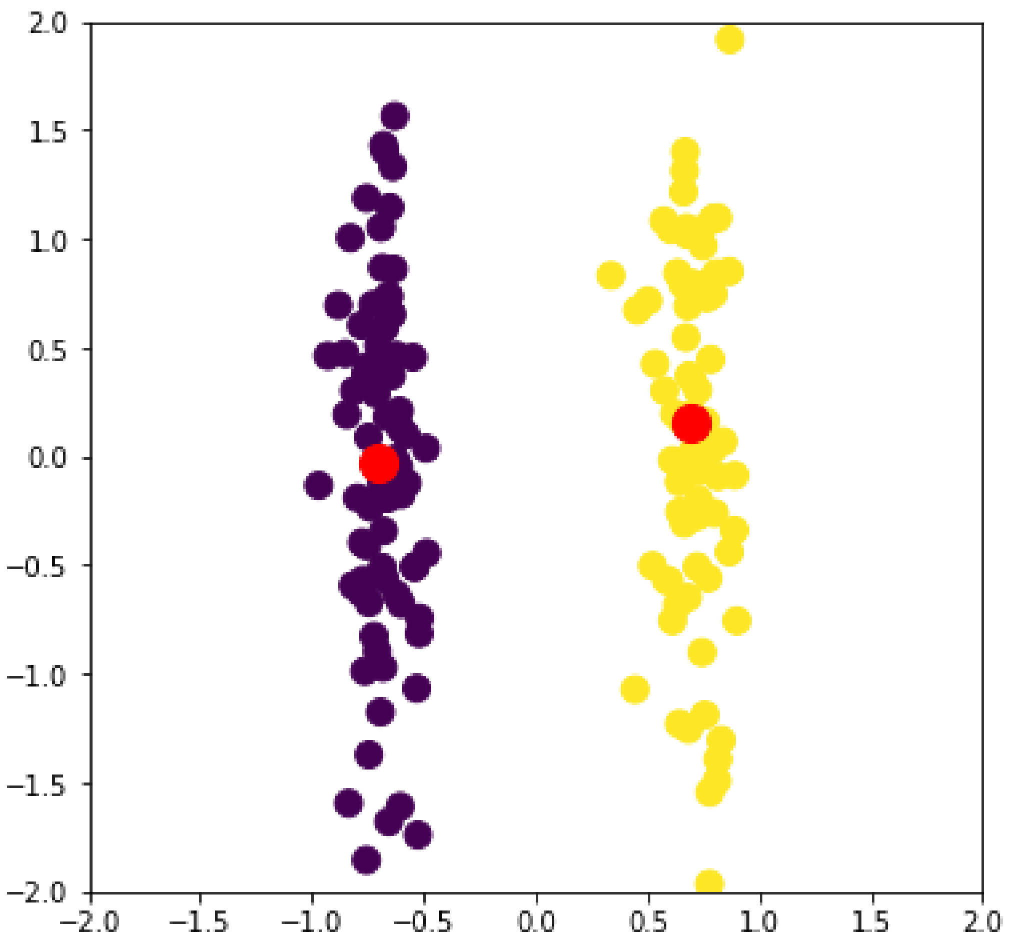
```
AMI= 0.008234430265824053
```



As we can see, the K-means prototypes are completely out of the data densities, The advantage of K-medoids is that the cluster prototypes will be actual examples in the data, so there is a better chance that the algoritm will be more resilient to different variances if clusters are enough separated.

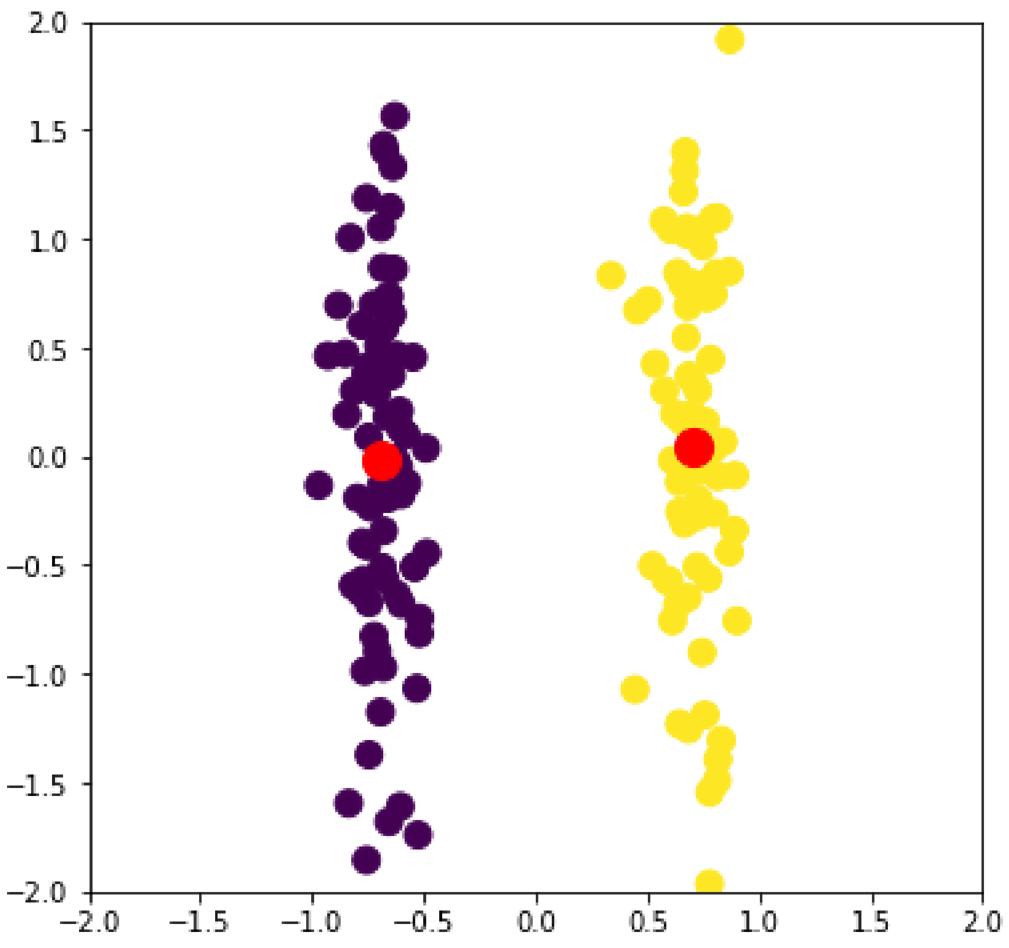
[21]:
```
kmd = KMedoidsFlexible(n_clusters=2, max_iter=200)
labels = kmd.fit_predict(data)
print("AMI=", adjusted_mutual_info_score(dlabels, labels))
plt.figure(figsize=(6,6))
plt.ylim(-2,2)
plt.xlim(-2,2)
plt.scatter(data[:, 0], data[:, 1], c=labels, s=100)
plt.scatter(kmd.cluster_medoids_[:,0], kmd.cluster_medoids_[:,1], c='r',␣
 ↪s=200);
```

```
AMI= 1.0
```



[22]:
```
gmm = GaussianMixture(n_components=2, covariance_type='diag')
gmm.fit(data)
labels = gmm.predict(data)
print("AMI=",adjusted_mutual_info_score(dlabels, labels))
plt.figure(figsize=(6,6))
plt.ylim(-2,2)
plt.xlim(-2,2)
plt.scatter(data[:, 0], data[:, 1], c=labels, s=100)
plt.scatter(gmm.means_[:,0], gmm.means_[:,1], c='r', s=200);
```

```
AMI= 1.0
```



The estimation of the parameters of the distribution done by GMM also has a better chance to be close to the data.

If you play with the characteristics of the dataset, you will see that usually if data are separated enough or the variances are different enough, clusters that have similar number of examples are partitioned right most of the time.

2.5.3 Density based clustering

Density based clustering does not assume a specific shape for the clusters or that the number of clusters is known. The goal is to uncover areas of high density in the space of examples. There are different strategies to find the dense areas of a dataset, but the usual methods are derived from the works of the algorithm DBSCAN [42]. Initially, it was defined for spatial databases, but it can be applied to data with more dimensions.

DBSCAN

This algorithm is based on the idea of *core points*, that constitute the examples that belong to the interior of the clusters, and the neighborhood relations of these points with the rest of the examples.

We define the set *ε-neighborhood*, as the instances that are at a distance less than ε to a given instance,

$$N_\varepsilon(x) = \{y \in X | d(x,y) \leq \varepsilon\} \tag{2.19}$$

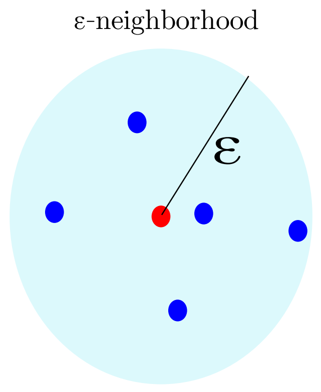


Figure 2.10: DBSCAN corepoints and ε-neighborhood

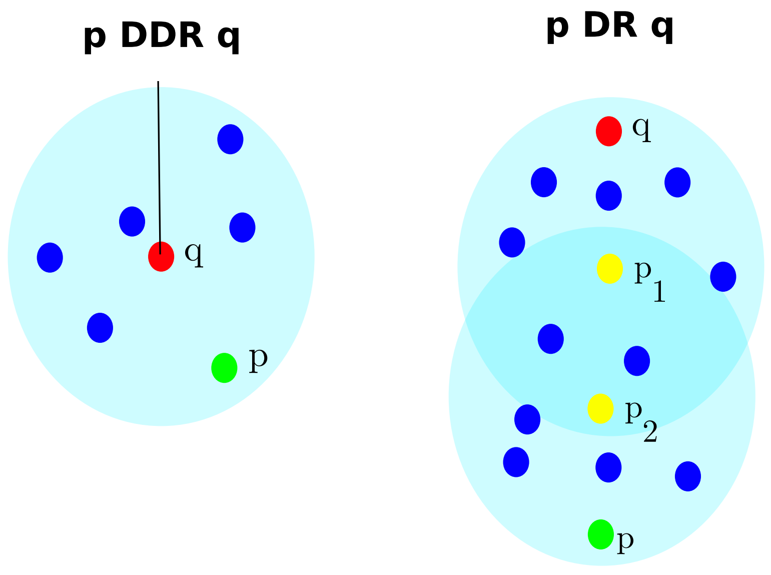


Figure 2.11: DBSCAN reachability relations

We define *core point* as the example that has a certain number of elements in $N_\varepsilon(x)$

$$Core\_point \equiv |N_\varepsilon(x)| \geq MinPts \tag{2.20}$$

From this neighborhood sets, different reachability relations are defined allowing to connect density areas defined by these core points. We say that two instances p and q are *Direct Density Reachable* with respect to ε and $MinPts$ if:

1. $p \in N_\varepsilon(q)$
2. $|N_\varepsilon(q)| \geq MinPts$

We say that two instances p and q are *Density Reachable* if there is a sequence of instances $p = p_1, p_2, \ldots, p_n = q$ where p_{i+1} is direct density reachable from p_i. And, finally, p and q are *Density connected* if there is an instance o such that both p and q are *Density Reachable* from o.

A cluster is defined as all the core points that are connected by these reachability relations, and the points that belong to their neighborhoods. Formally, given a dataset D, a cluster C with respect ε and $MinPts$ is any subset of D that:

1. $\forall p, q \; p \in C \wedge density\_reachable(q, p) \longrightarrow q \in C$
2. $\forall p, q \in C \; density\_connected(p, q)$

Any point that can not be connected using these relationships is treated as noise.

This is an outline of the DBSCAN algorithm:

1. Start with an arbitrary instance and compute all density reachable instances with respect to a given ε and $MinPts$.
2. If it is a core point, we will gather all examples related to this example to obtain a group (all these examples are labeled and discarded from further consideration), otherwise, the instance belongs to the border of a cluster or is an outlier, so it is not considered as starting point.
3. Repeat the first step until no more core points are found.
4. Label the non clustered examples as noise.

Figure 2.12 shows an example of the *ε-neighborhood* of a set of data points, all of those that have more than a number of data points inside form clusters.

The key point of this algorithm is the choice of the ε and $MinPts$ parameters. A possibility is to set heuristically their values using the thinnest cluster from the dataset.

As an extension of DBSCAN, the algorithm OPTICS [4] uses heuristics based on the histogram of the distances among the examples to find good values for these parameters. The idea is to compute two distances, the *core distance* of an example, that is the smallest distance of the example to the closest example in its ε neighborhood (this is undefined if the example is not a core point) and the *reachability distance* between two examples p and o that is the smallest distance such the p example is directly density reachable from o (this is undefined if o is not a core point). This information is used to find out what clusters can be obtained for any value less than a given ε.

This also allows to order the examples for each cluster, and their reachability distance to obtain an idea about the clusters and their densities. This can be represented as a reachability plot, that can be used to tune the ε. It can also be used to define the border between clusters to select automatically what examples should be considered as outliers using a parameter ξ that defines how much must change the reachability distance between two consecutive examples to consider that we are outside the cluster.

The main drawback of these methods come from the cost of finding the nearest neighbors for an example. To decrease the computational cost a R* tree, or a ball tree can be used to index all examples, but these structures degrade with the number of dimensions to a linear search. This makes the computational time of these algorithms proportional to the square of the number of examples for datasets with many dimensions.

DENCLUE

Other example of density based clustering algorithm is DENCLUE ([64]). It is based on kernel density estimation, and it defines the influence of an example in a dataset as the sum of the densities based on the example computed, using a kernel for all the dataset (for instance a gaussian kernel):

$$f_B^y(x) = f_B(x, y) = \sum_{i=1}^{N} e^{-\frac{1}{2}\frac{|x-y_i|^2}{\sigma^2}} \tag{2.21}$$

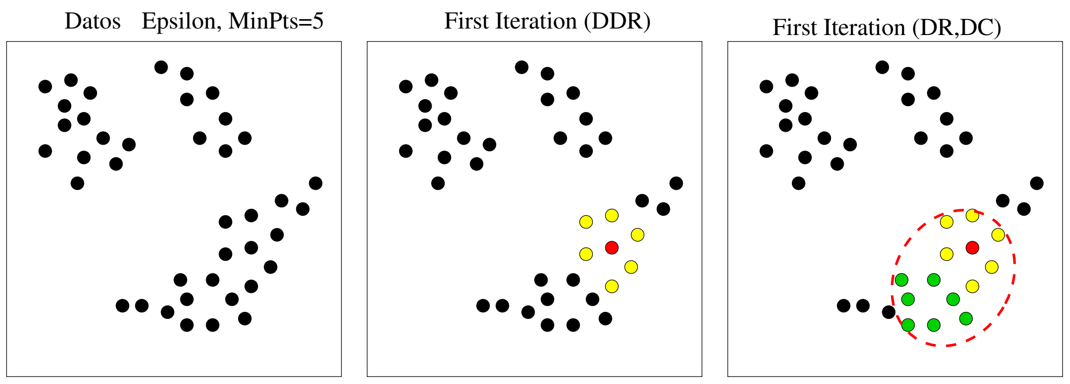


Figure 2.12: Example of the DBSCAN algorithm

It defines the density function of a dataset as the sum of the influences of all examples of the dataset

$$f_B^D(x) = \sum_{i=1}^{N} f_B^{x_i}(x_i, x) \tag{2.22}$$

This function takes the role of the ε-neighborhood from DBSCAN, but in this case as a continuous function. Instead of using density relations among points, it defines the gradient of $f_B^D(x)$ as:

$$\bigtriangledown f_B^D(x) = \sum_{i=1}^{N} (x_i - x) f_B^{x_i}(x_i, x) \tag{2.23}$$

so the direction of the gradient points to the more dense parts of the datasets, assimilated to the core points in DBSCAN.

The algorithm defines that a point x^* is a *density-attractor* iff is a local maximum of the density function f_B^D. A point x is *density-attracted* to a density-attractor iff $\exists k \in N; d(x^k, x^*) \leq \epsilon$. This defines a path that connects the points of a dataset to the density attractors based on the density gradient function:

$$x^0 = x, x^i = x^{i-1} + \delta \cdot \frac{\bigtriangledown f_B^D(x^{i-1})}{\| \bigtriangledown f_B^D(x^{i-1}) \|} \tag{2.24}$$

This can be assimilated to the different density relationships defined by DBSCAN.

Two kinds of clusters can be defined, *Center-defined Cluster* (wrt to σ,ξ) for a density-attractor x^* as the subset of examples being density-attracted by x^* and with $f_B^D(x^*) > \xi$. This bias the algorithm towards finding spherical clusters centered on the more dense areas of a dataset defined by a central point. We have also an *Arbitrary-shape cluster* (wrt to σ,ξ) for a set of density-attractors X as the subset of examples being density-attracted to any $x^* \in X$ with $f_B^D(x^*) > \xi$ and with any density-attractor from X connected by a path P with $\forall p \in P : f_B^D(p) > \xi$. This allows obtaining a cluster as the sum of different close dense areas. Figure 2.13 show an example of the gradients of the density function and the central points of the densities as density attractors.

This algorithm is more flexible than DBSCAN because of the use of the approximation of the densities using kernel density estimation. This makes that different algorithms can be reproduced with the right choice of σ and ξ parameters, for example the own DBSCAN can be obtained using arbitrary shaped clusters, K-means using center defined clusters and hierarchical clustering merging different densities hierarchically.

The algorithm is divided in two phases:

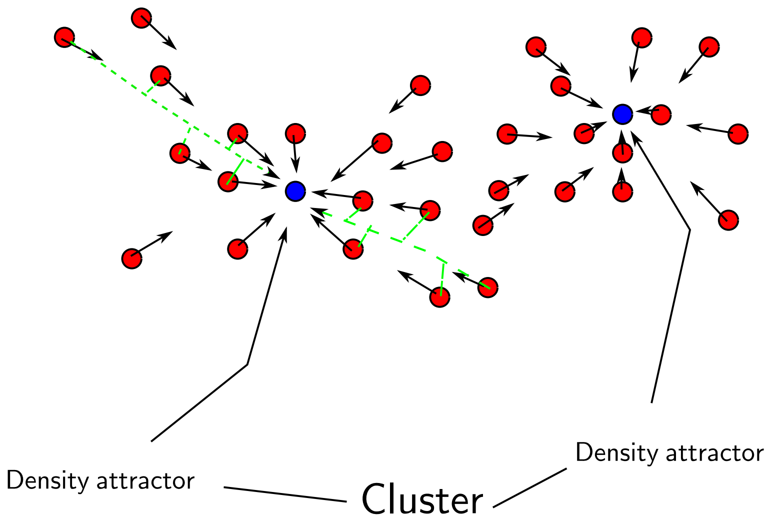


Figure 2.13: DENCLUE algorithm

1. *Preclustering:* The dataspace is divided in d-dimensional hypercubes, only using the cubes with datapoints. These hypercubes are mapped to a tree structure for efficient search. From these hypercubes, given a threshold, only the highly populated ones, and their neighbors are considered.

2. *Clustering:* For each datapoint in the hypercubes a local density function, and a local gradient are computed. Using a Hill-Climbing algorithm the density-attractor for each point is computed. For efficiency reasons each point near the path computed during the search of a density-attractor is assigned to that density-attractor.

   Due to efficiency reasons, the density functions are only computed for the neighbors of each point.

2.5.4 Grid based clustering

Grid based clustering is another approach to finding dense areas of examples. The basic idea is to divide the space of instances in hyper-rectangular cells by discretizing the attributes of the dataset. In order to avoid generating a combinatorial number of cells, different strategies are used. It has to be noticed the fact that, the maximum number of cells that contains any example is bounded by the size of the dataset.

These methods have the advantage of being able to discover clusters of arbitrary shapes and also the number of cluster has not to be decided beforehand. The different algorithms usually rely on some hierarchical strategy to build the grid top-down or bottom-up.

STING

The algorithm STING [134] assumes that the data has a spatial relationship (usually two-dimensional), and beginning with one cell, recursively partitions the current level into four cells obtaining a hierarchical grid structure determined by the density of the examples (see figure 2.14). Each level divides the cells from the previous level in rectangular areas, and the size of the cells depends on the density of the examples. Each cell is summarized by the sufficient statistics of the examples it contains (number of instances, mean, deviation, min value, max value, type of distribution).

This structure allows querying for regions that hold certain conditions. The structure is traversed, and the cells containing data relevant to the query are returned. The algorithm is as

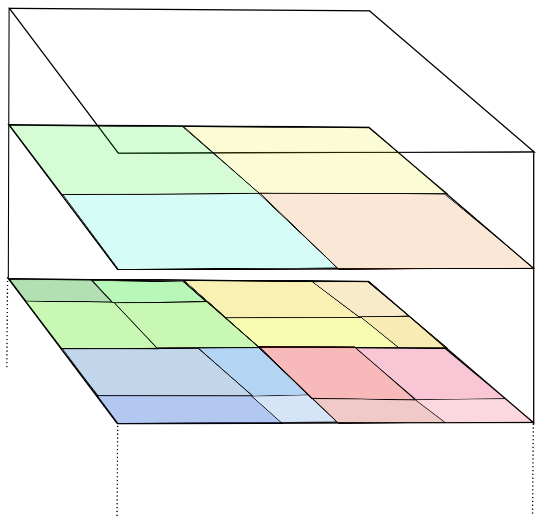


Figure 2.14: Hierarchical grid structure built by STING

follows:

1. For each cell of a layer determine if it is relevant to the query
2. If it is not the bottom layer repeat the process with the cells marked relevant
3. If it is the bottom layer, find all the relevant cells that are connected and return the region that they form

The results of this algorithm approximate those returned by DBSCAN as the granularity approaches zero.

CLIQUE

The algorithm of CLIQUE [3] uses a more general approach. It assumes that the attributes of the dataset have been discretized, and the one dimensional dense cells for each attribute can be identified. These cells are merged attribute by attribute in a bottom up fashion, considering that a merging only can be dense if the cells of the attributes that compose the merger are dense. This antimonotonic property allows to prune the space of possible cells. Once the cells are identified, the clusters are formed by finding the connected components in the graph defined by the adjacency relations of the cells.

The algorithm can generate CNF descriptions from the groups that discovers and its goal is to find a space with fewer dimensions where the groups are easier to identify. The process is divided in three steps:

1. Identify the subspaces with clusters:

   A bottom up strategy is applied identifying first the bins of higher density in one dimension and then combining them (see figure 2.15). A combination of $k+1$ dimensions can only have high density bins if there are high density bins in k dimensions, this rule gives the set of candidates to high density bins when we increase the dimensionality by one. Some other heuristics are also used to reduce the computational cost of the search

2. Identify the clusters:

   A set of dense bins are received from the previous step, the contiguous bins are considered as forming part of the same cluster (see figure 2.16). Two bins are connected if they share a

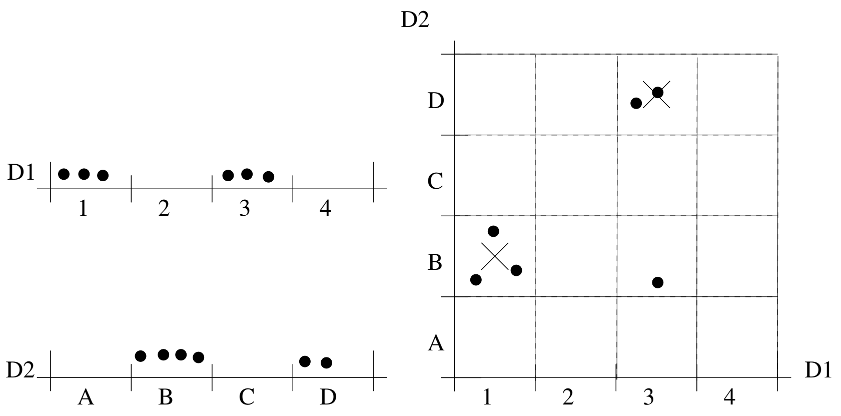


Figure 2.15: CLIQUE algorithm step 1

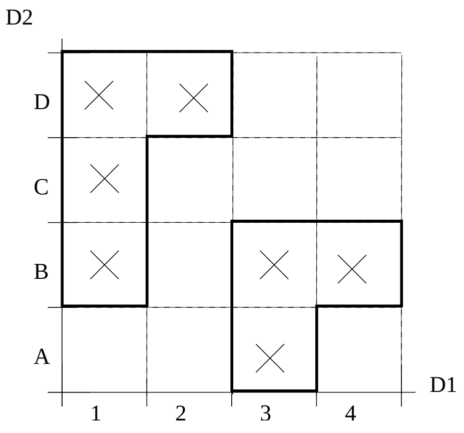


Figure 2.16: CLIQUE algorithm step 2

side in one dimension or there is an intermediate bin that connects them. This problem is equivalent to finding the connected components of a graph.

3. Generate the minimal description of the clusters:

   Now a set of connected components in k dimensions are received, and the problem to solve is to look for the minimal cover for each connected component (see figure 2.17). Being this problem NP-hard, an approximation is computed using rectangles that cover part of the group maximally and after this the redundancies are reduced, obtaining a computational affordable approximation to the problem.

All these methods can usually scale well, but it depends on the granularity of the discretization of the space of examples. The strategies used to prune the search space allow reducing largely the computational cost, scaling linearly on the number of examples, and a quadratical factor in the number of attributes.

2.6 Notebook: Density Based Clustering

Density based clustering does not assumes an specific shape for the data, let's see wen we use data that does not fit into spherical clusters, for example an image

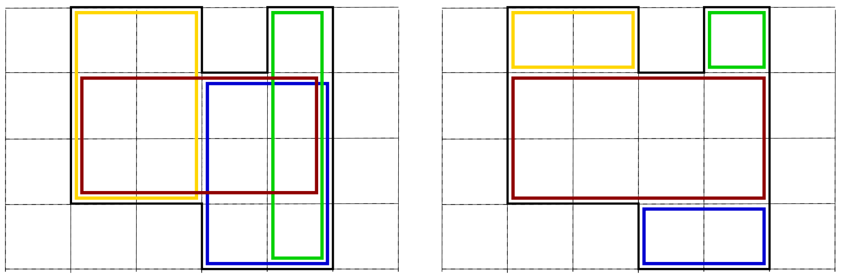
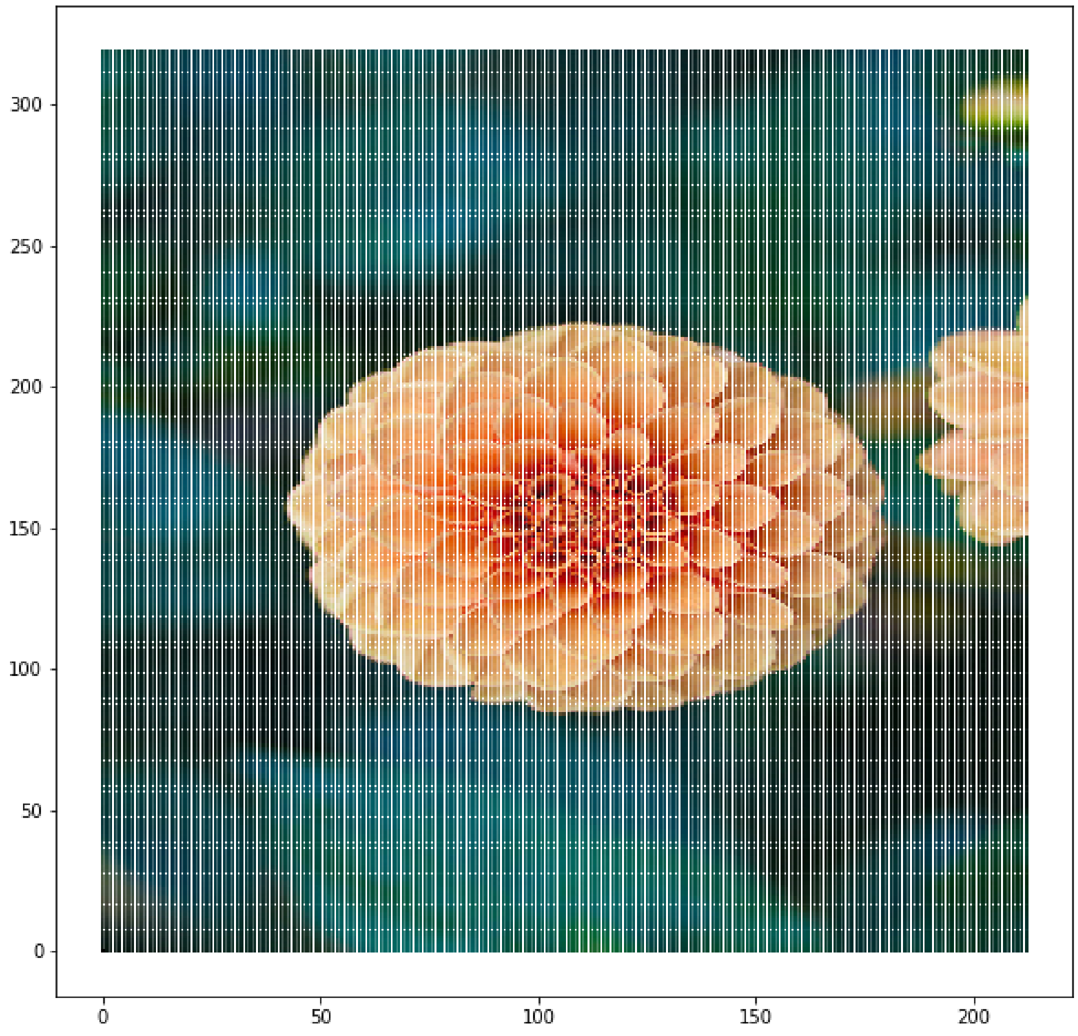


Figure 2.17: CLIQUE algorithm step 3

[1]:
```python
from sklearn.datasets import load_sample_image
from sklearn.cluster import DBSCAN, KMeans
import matplotlib.pyplot as plt
import numpy as np

import warnings
warnings.filterwarnings('ignore')
```

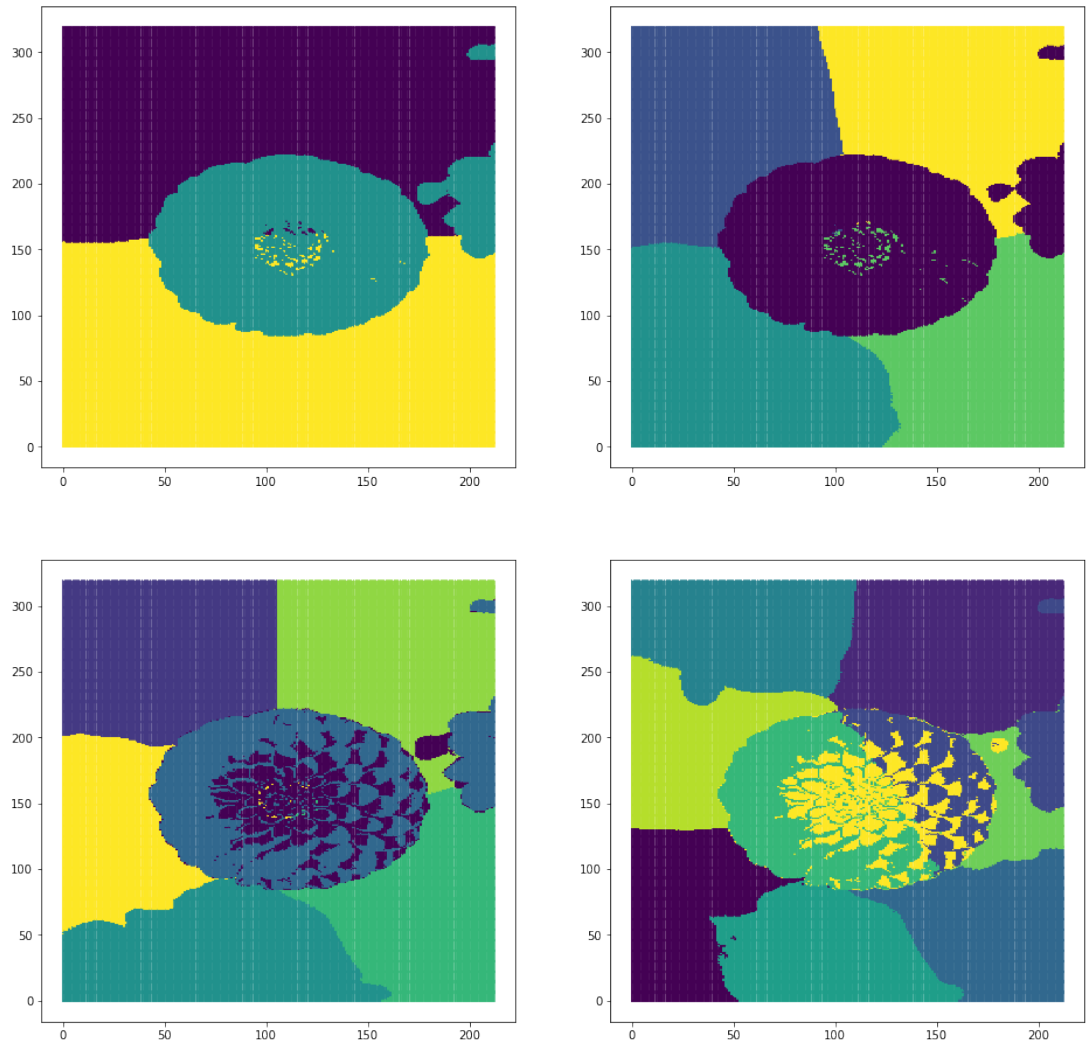
We will transform the image into a dataset including the coordinates of the pixels and their RGB values

[2]:
```python
colors = 'rgbymc'
# you can use 'flower' or 'china' images
flower = load_sample_image('flower.jpg')
mdata = np.zeros((int(flower.shape[0]*flower.shape[1]/4.0), flower.
 ↪shape[2]+2))
cc=0
for i in range(0,flower.shape[0]-1,2):
    for j in range(0,flower.shape[1]-1,2):
        mdata[cc][0] = i/2
        mdata[cc][1] = j/2
        for k in range(flower.shape[2]):
            mdata[cc][2+k] = flower[i, j, k]
        cc += 1
plt.figure(figsize=(10,10))
plt.scatter(mdata[:, 0], mdata[:, 1], c=mdata[:, 2:]/255.0, s=2, marker='+');
```



This is what happens using K-means looking for different number of cluster using the coordinates and the color

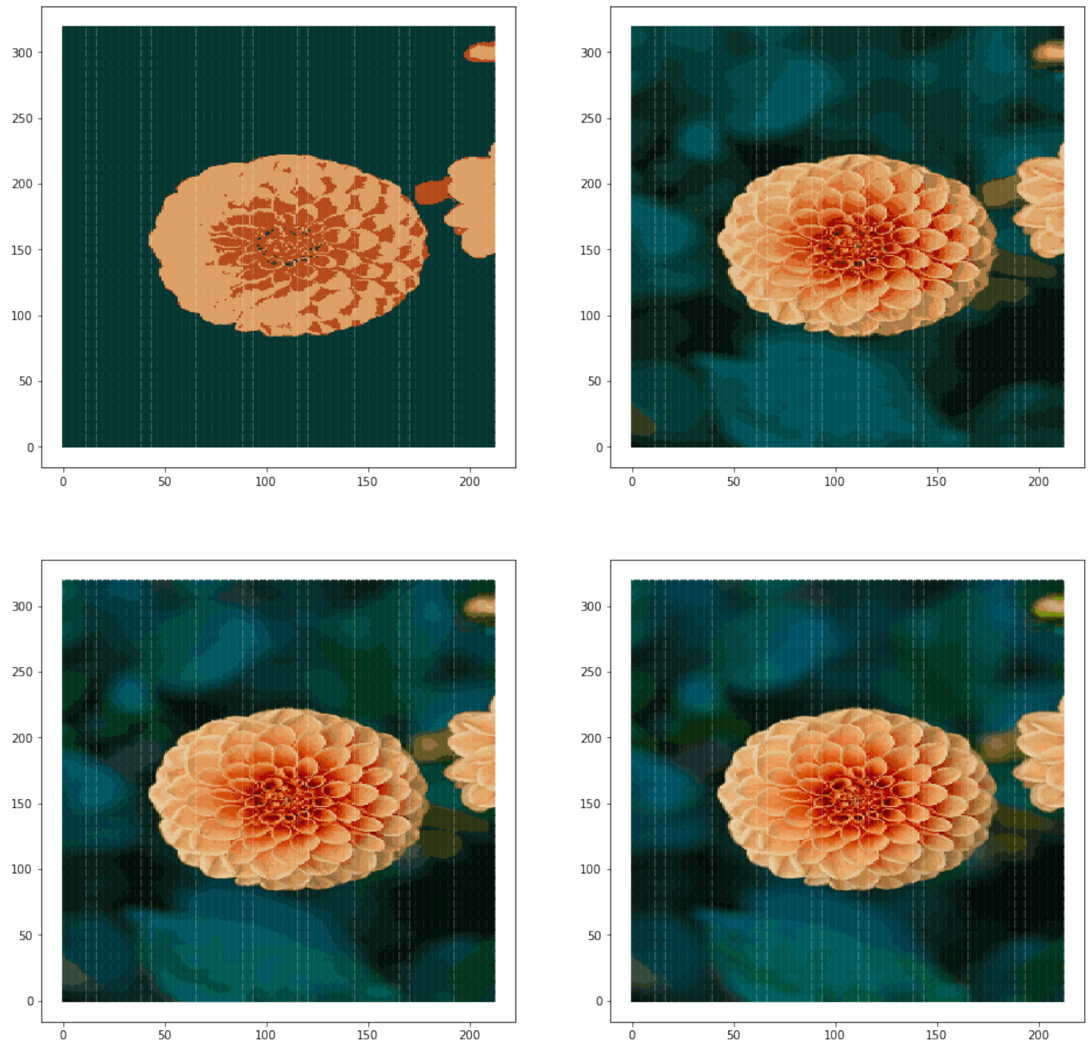
[3]:
```
lnc=[3,5,7,10]
fig = plt.figure(figsize=(16,16))
for i, nc in enumerate(lnc):
    km = KMeans(n_clusters=nc, n_jobs=-1)
    labels = km.fit_predict(mdata)
    ax = fig.add_subplot(2,2,i+1)
    plt.scatter(mdata[:, 0], mdata[:, 1], c=np.array(labels)/len(np.
 ↪unique(labels)), s=2, marker='+');
```



Even when we do not have actual spherical clusters we can find similar pixels around regions that make sense for K-means

Now we will cluster only the colors (we are applying vector quantization to the colors palette). You can play with the number of clusters to see how the cluster colors get closer to the original colors.

[4]:
```
lnc=[3,15,30,60]
fig = plt.figure(figsize=(16,16))
for i, nc in enumerate(lnc):
    km = KMeans(n_clusters=nc)
    labels = km.fit_predict(mdata[:,2:])
    ccent = km.cluster_centers_/255.0
    lcols = [ccent[l,:] for l in labels]
    ax = fig.add_subplot(2,2,i+1)
    plt.scatter(mdata[:, 0], mdata[:, 1], c=lcols, s=2, marker='+');
```

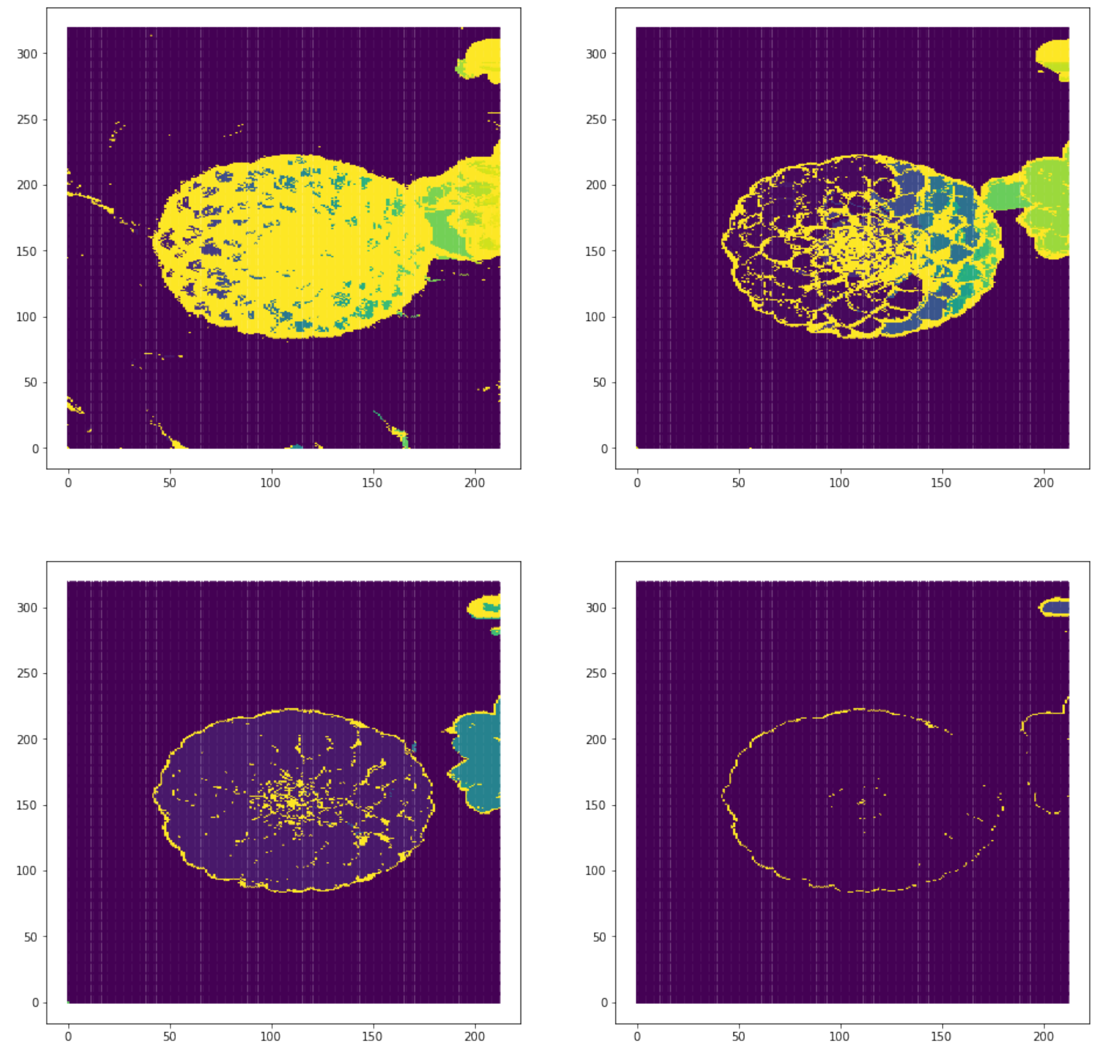


Even when the number of colors in the image is large, clustering the RGB values can compress the color information of the image enough to not to appreciate the difference when the number of clusters is large enough

For DBSCAN we will also use coordinates and colors, now to select the parameters are more complex and different values will yield very different results, so it is important to be able to visualize the clusters.

```
[5]: lpar = [(5, 5),(10, 10), (15, 15),(25, 25)]

     fig = plt.figure(figsize=(16,16))
     for i, (eps, ms) in enumerate(lpar):
         dbs = DBSCAN(eps=eps, min_samples=ms)
         labels = dbs.fit_predict(mdata)
         unq = len(np.unique(labels))
         print(f"NClusters(eps={eps},ms={ms})={unq-1}")
         ecolors = np.array(labels)
         ecolors[ecolors == -1] += unq+3
         ax = fig.add_subplot(2,2,i+1)
         plt.scatter(mdata[:, 0], mdata[:, 1], c=ecolors, s=2, marker='+');
```

```
NClusters(eps=5,ms=5)=324
NClusters(eps=10,ms=10)=84
NClusters(eps=15,ms=15)=13
NClusters(eps=25,ms=25)=2
```



As you can see, very different number of clusters are obtained changing the parameters of the algorithm.

2.6.1 Other clustering approaches

There are several other approaches that use other methodologies for obtaining a set of clusters from a dataset. We are going to outline three different approaches: methods based on graph theory and properties of neighborhood graphs, methods based on unsupervised neural networks and methods based on support vector machines.

Graph based methods

There are some classical approaches to graphs clustering that use basic principles as criteria for partitioning the data and that are the base for more complex algorithm. The idea is that different kinds of graphs can be computed using the similarity matrix defined by the examples, such

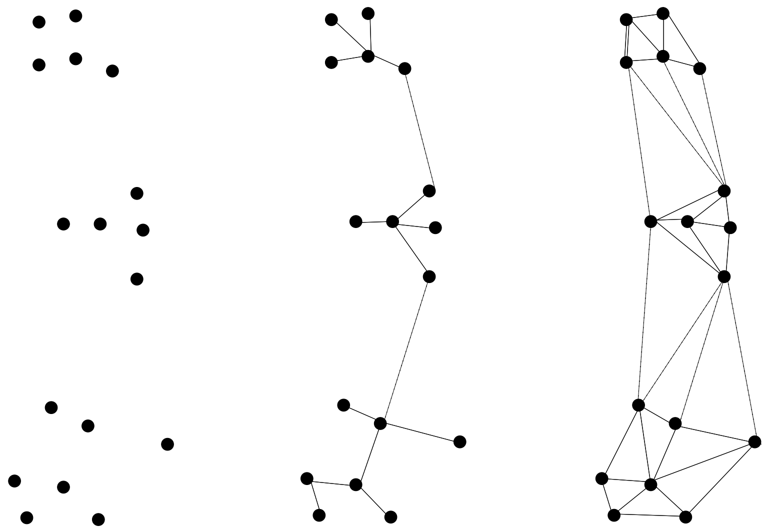


Figure 2.18: Different graphs (MST/Delanau) computed from a set of examples

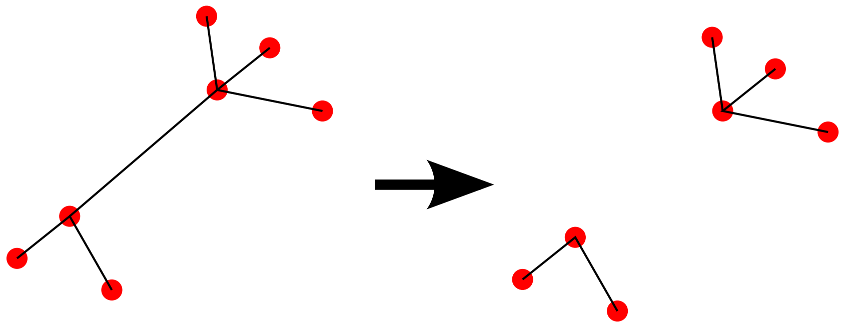


Figure 2.19: Deleting inconsistent edges from an examples graph generates clusters

as the Minimum Spanning Tree (MST), Voronoi tesselation or Delanau triangularization (see figure 2.18) that capture at different levels the local structure of the data. Using these graphs, a consistency criteria for the edges of the graph is defined.

This criterion can be used agglomeratively or divisively, merging examples/groups that have edges below a threshold or deleting inconsistent edges above a threshold until some unconnected component are obtained from the initial graph (see figure 2.19). These components define the clusters of the data. These algorithms have two advantages, first we do not need to define the number of clusters beforehand, this number is controlled by the consistency criteria and the densities in the data. Second, we do not impose a specific model for the clusters, so they can have any shape, being more adaptive to the actual densities.

An example of this methodology is CHAMELEON ([78]). This is an agglomerative algorithm that is based on the graph obtained from the matrix distance using only the k-nearest neighbors of each example. It defines two measures for the decision about the consistency of the edges of the graph, the relative interconnectivity $RI(C_i, C_j)$ and relative closeness $RC(C_i, C_j)$ between subgraphs. These measures are defined from the cost of the edges that divide two groups. Two groups could be merged if the interconnectivity between their graphs, and the closeness among instances is high.

- Relative Interconnectiviry ($RI(C_i, C_j)$): the sum of the edges that connect the two groups divided by the sum of the edges that partitions each group in two equal sized groups
- Relative Closeness ($RC(C_i, C_j)$): the mean of the edges that connect the two groups divided

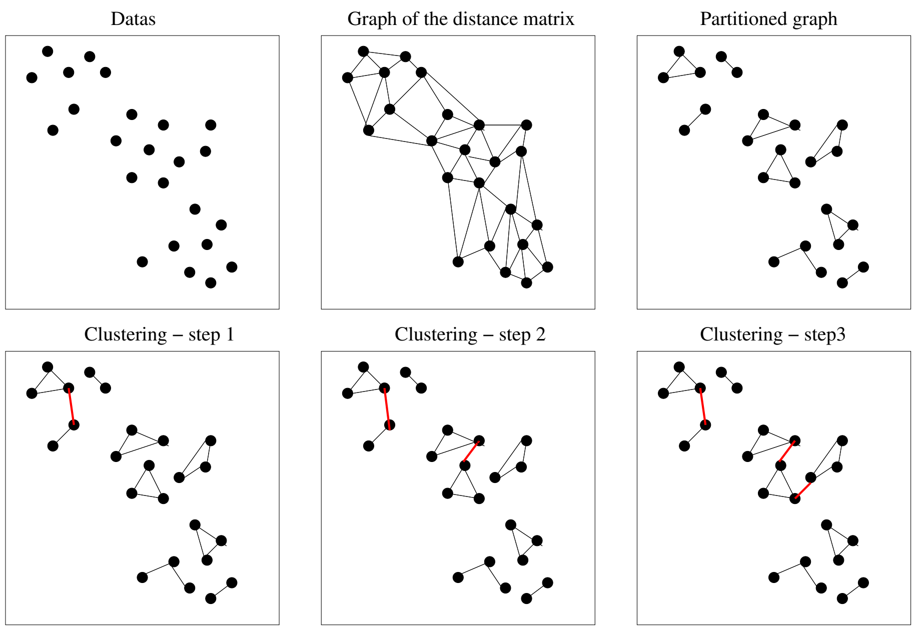


Figure 2.20: Example of the CHAMALEON algorithm in action. First the k-neighbors graph is defined and the graph is partitioned so small components are obtained, then new edges are added in order optimizing the RI/RC combination measure.

by the mean of the edges that partitions each group in two equal sized groups

Initially the algorithm divides the graph generated by deleting edges from the distance matrix until many small groups are obtained (minimizing the cost of the edges inside the group). Then it uses an agglomerative process to merge the groups using the measures of relative interconnectivity and closeness ($RI(C_i, C_j) \cdot RC(C_i, C_j)^\alpha$ is used as goal function). Figure 2.20 presents an example of how this algorithm works.

Spectral Clustering

Spectral graph theory defines properties that hold the eigenvalues and eigenvectors of the adjacency matrix or Laplacian matrix of a graph. Spectral clustering [88] uses the Laplacian matrix defined from the similarity matrix of the dataset. Different clustering algorithms can be defined depending on how it is computed. For instance, the whole matrix can be used or only the neighborhood graph. Also, different kinds of normalization can be applied to the Laplacian matrix.

If we start with the similarity matrix (W) of a dataset (complete or only k-neighbors), the degree of an edge is defined as:

$$d_i = \sum_{j=1}^{n} w_{ij}$$

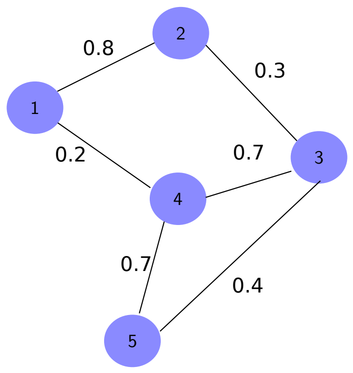
The degree matrix D is defined as the matrix with values $d_1, d_2, \ldots, d_n$ as diagonal. From this matrix different Laplace matrices can be computed:

- Unnormalized: $L = D - W$
- Normalized: $L_{sym} = D^{-1/2} L D^{-1/2}$ or also $L_{rw} = D^{-1} L$

Assuming that the Laplacian matrix represents the neighborhood relationships among examples, applying an eigen decomposition of this matrix, the eigenvectors of this matrix define a new set of coordinates that can be interpreted as a dimensionality reduction method if only the k components corresponding to the k highest eigen values are used.

It is known that if there are well-defined clusters in the data, the decomposition of the Laplacian matrix will have only a specific number of eigenvalues larger than zero, corresponding to the number of clusters. Also, the eigenvectors are in a space where the clusters are easier to identify. This means that traditional clustering algorithms can be applied to discover these clusters, for instance, K-means.

For instance, given the following graph representing the similarity among examples:



We have the following affinity matrix, degree matrix and Laplacian matrix:

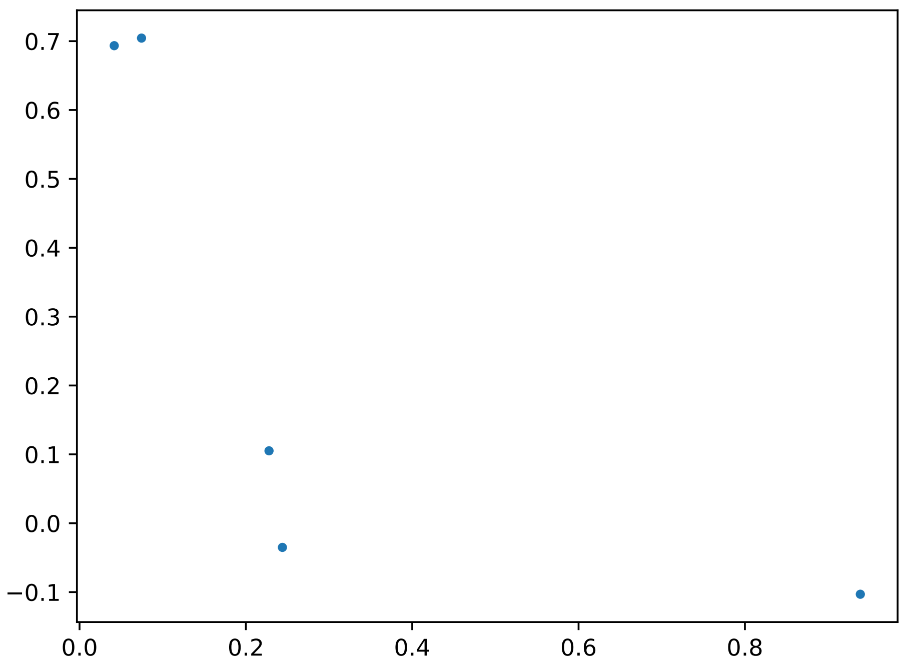
$$W = \begin{pmatrix} 0 & 0.8 & 0 & 0.2 & 0 \\ 0.8 & 0 & 0.3 & 0 & 0 \\ 0 & 0.3 & 0 & 0.7 & 0.4 \\ 0.2 & 0 & 0.7 & 0 & 0.7 \\ 0 & 0 & 0.4 & 0.7 & 0 \end{pmatrix} D = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1.1 & 0 & 0 & 0 \\ 0 & 0 & 1.4 & 0 & 0 \\ 0 & 0 & 0 & 1.6 & 0 \\ 0 & 0 & 0 & 0 & 1.1 \end{pmatrix}$$

$$L = \begin{pmatrix} 1 & -0.8 & 0 & -0.2 & 0 \\ -0.8 & 1.1 & -0.3 & 0 & 0 \\ 0 & -0.3 & 1.4 & -0.7 & -0.4 \\ -0.2 & 0 & -0.7 & 1.6 & -0.7 \\ 0 & 0 & -0.4 & -0.7 & 1.1 \end{pmatrix}$$

After eigen decomposition we have as first two components of the eigenvalues:

$$Eigvec(1,2) = \begin{pmatrix} 0.07 & 0.70 \\ 0.04 & 0.69 \\ 0.22 & 0.10 \\ 0.93 & -0.10 \\ 0.24 & 0.03 \end{pmatrix}$$

That maps the graph in two dimensions as follows:



Also, graph partitioning algorithms can be applied to the matrix for obtaining a set of unconnected components. For instance, the min-cut problem can be applied to the defined graph. Several objective functions have been defined for this purpose. Given two disjoint sets of vertex A and B, we define:

$$cut(A,B) = \sum_{i \in A, j \in B} w_{ij} \tag{2.25}$$

We can partition the graph solving the mincut problem choosing a partition that minimizes:

$$cut(A_1,\ldots,A_k) = \sum_{i=1}^{k} cut(A_i,\overline{A_i}) \tag{2.26}$$

Sometimes, using directly the weights of the Laplacian does not always obtain a good result. There are alternative objective functions that are also used like:

$$RatioCut(A_1,\ldots,A_k) = \sum_{i=1}^{k} \frac{cut(A_i,\overline{A_i})}{|A_i|} \tag{2.27}$$

$$Ncut(A_1,\ldots,A_k) = \sum_{i=1}^{k} \frac{cut(A_i,\overline{A_i})}{vol(A_i)} \tag{2.28}$$

where $|A_i|$ is the size of the partition and $vol(A_i)$ is the sum of the degrees of the vertex in A_i.

The computational complexity of this family of methods is usually high because the computation of the eigenvectors of the Laplacian matrix is needed. This cost can be reduced by using approximate methods for estimating the first k eigenvalues of the matrix.

Affinity clustering

Affinity clustering [48] is an algorithm based on message passing but related to graph partitioning. It also uses the graph obtained from the data similarity matrix. The algorithm computes the set of examples that have to be the cluster prototypes and how the examples are assigned to them. Each pair of objects have a distance defined $s(i,k)$ (e.g.: euclidean distance). The number of clusters is not fix a priori, but depends on a parameter called *damping factor* that controls how the information about the assignment of examples to clusters is computed.

The method defines two measures, *responsibility*, that accounts for the suitability of an exemplar for being a representative of a cluster and *availability*, that accounts for the evidence that certain point is the representative of other example.

- *Responsibility* $r(i,k)$, that is a message that an example i passes to the candidate to cluster exemplar k of the point. This represents the evidence of how good is k for being the exemplar of i
- *Availability* $a(i,k)$, sent from candidate to cluster exemplar k to point i. This represents the accumulated evidence of how appropriate would be for point i to choose point k as its exemplar

These measures are initialized using the similarity among the examples, and also, each example begins with a value for $r(k,k)$ that represents the preference for each point to be an exemplar. Iteratively, the number of clusters, and their representatives are determined by refining these measures, that are linked by the following set of equations:

- All availabilities are initialized to 0
- The responsibilities are updated as:
$$r(i,k) = s(i,k) - max_{k' \neq k}\{a(i,k') + s(i,k')\} \tag{2.29}$$
- The availabilities are updated as:
$$a(i,k) = min\{0, r(k,k) + \sum_{i' \notin \{i,k\}} max(0, r(i',k))\} \tag{2.30}$$
- The self availability $a(k,k)$ is updated as:
$$a(k,k) = \sum_{i' \neq k} max(0, r(i',k))\} \tag{2.31}$$
- The exemplar for a point is identified by the point that maximizes $a(i,k) + r(i,k)$, if this point is the same point, then it is an exemplar.

The algorithm proceeds iteratively as follows:

1. Update the responsibilities given the availabilities
2. Update the availabilities given the responsibilities
3. Compute the exemplars
4. Terminate if the exemplars do not change in a number of iterations

See figure 2.21 for an example of this algorithm in action. The temporal computational complexity of this method is quadratic on the number of examples, and the space complexity is also quadratic, making it not suitable for large datasets. This algorithm is closely related to other message passing algorithms used for belief propagation in probabilistic graphical models

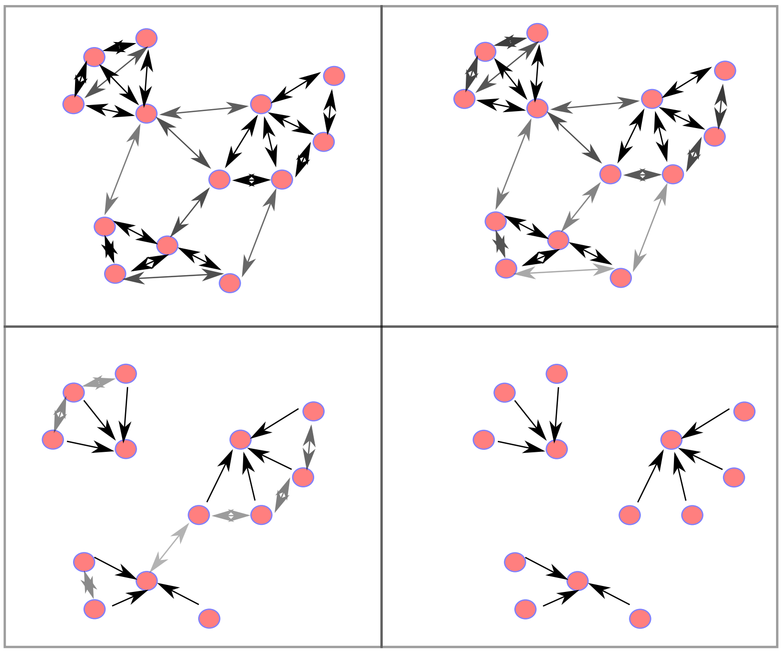


Figure 2.21: Affinity propagation in action. Each iteration the responsibility increases for some examples (cluster prototypes) and decreases for the rest.

Unsupervised neural networks

There are some unsupervised neural networks methods, among them *self-organizing maps* are widely used for visualization tasks. This method can be interpreted as an on-line constrained version of K-means. It transforms the data to fit in a 1-d or 2-d regular mesh (rectangular or hexagonal) that represents the clusters in the data. The nodes of this mesh correspond to the prototypes of the clusters. This algorithm can also be used as a dimensionality reduction method (from N to 2 dimensions).

To build the map we have to fix the size and shape of the mesh (rectangular/hexagonal). This constrains the number of clusters that can be obtained and how are they related. Each node of the mesh is a multidimensional prototype of p features.

The canonical SOM algorithm is outlined in algorithm 2.6. There can be seen some differences to the standard batch K-means algorithm. Each iteration, each example is assigned sequentially to the closer prototype, and it is moved a fraction in the direction of the example (instead of computing the centroid). Also, the neighbor prototypes are also affected. The effect of this procedure is to transform the low dimensional mesh to be closer to the data, but maintaining the fixed neighborhood relationships among the prototypes.

This algorithm has different variations that affect to the update strategy and the behavior of the parameters. The performance of the algorithm can be controlled by the learning rate α, that represents how much the prototype is moved towards the data each iteration. It is usually decreased from 1 to 0 during the iterations to accelerate convergence. The neighborhood of a prototype is defined by the adjacency of the cells and the distance among the prototypes. The number of neighbors used in the update can also be decreased during the iterations from a predefined number to 1 (only the prototype nearest to the observation). Also, the algorithm could give different weights to the effect on the neighbors depending on the distance among the prototypes, so if a prototype gets apart from the rest it will have less or no effect on them.

Figure 2.22 shows an example of execution of the algorithm. The process starts with a square 3×3 grid of prototypes and deforms the grid, so the prototypes are closer to the points. Finally, each cell of the grid is colored depending on the color of the original examples.

Algorithm 2.6 SOM algorithm

```
Algorithm: Self-Organizing Map algorithm
Initial prototypes are distributed regularly on the mesh
for Predefined number of iterations do
    foreach Example $x_i$ do
        Find the nearest prototype ($m_j$)
        Determine the neighborhood of $m_j$ ($\mathcal{M}$)
        foreach Prototype $m_k \in \mathcal{M}$ do
            $m_k = m_k + \alpha(x_i - m_k)$
        end
    end
end
```

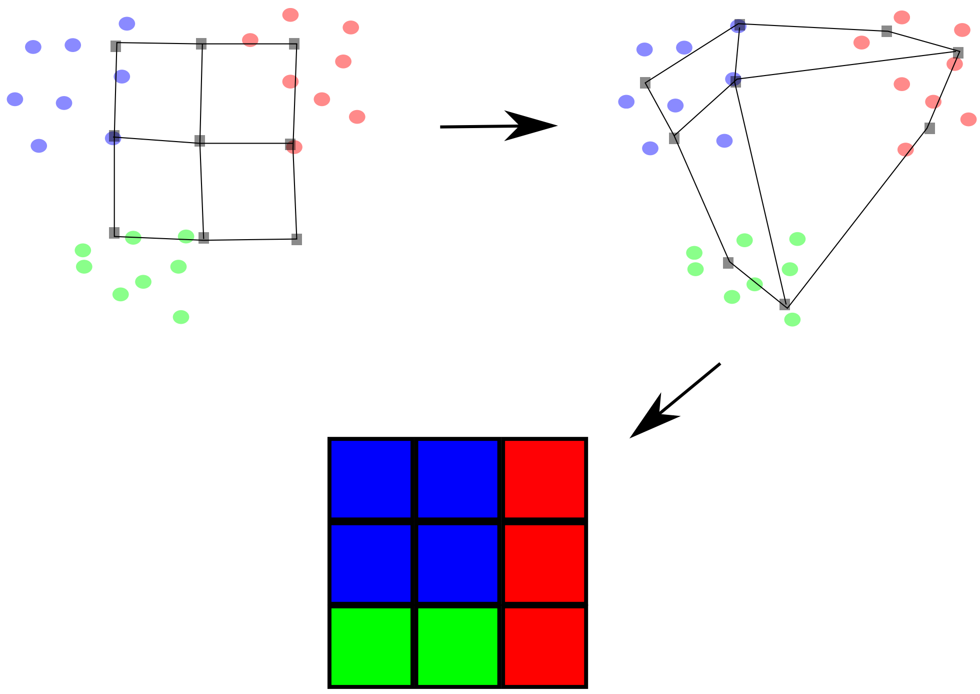


Figure 2.22: Example of self-organizing map using a 3×3 grid

SVM Clustering

Support Vector Machines are a supervised learning method that use kernel transformation of the features of the data for obtaining a separating hyperplane. This procedure can be extended in different ways to the unsupervised counterpart.

The simplest formulation is the approach using the one class-SVMs. These machines are able to work with only positive examples and can be used to estimate their probability density. They are usually applied to outlier detection, so examples that are outside the separating hyperplane are considered negative examples.

This allows defining different algorithms, the original one was defined in [12], that finds the smallest enclosing sphere in feature space for the data using a gaussian kernel:

$$||\Phi(x_j) - a||^2 \leq R^2 \;\; \forall j \tag{2.32}$$

with a the center of the sphere and R the minimum radius. The data is then mapped back to the original space, and the support vectors (examples that lie in the border of the sphere) are

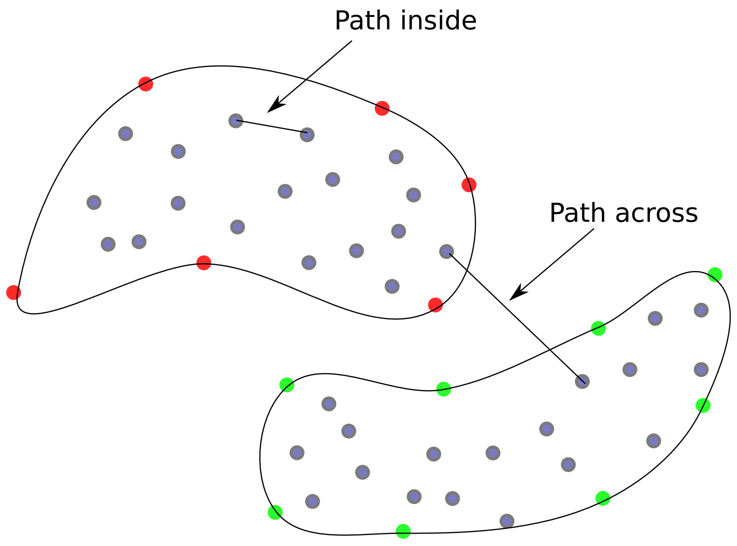


Figure 2.23: Examples of SVM clustering. The labeling of the examples is determined by determining the sphere that contains each example

used to partition the dataset. The labeling of the examples is determined assuming that the sphere is divided in different spheres in the original space and, the path that connects examples from different clusters has to cross outside the sphere borders (see figure 2.23).

Other approaches use an algorithm similar K-means like [17] or define a more general optimization problem that looks for a set of dividing planes for the examples like Maximum Margin Clustering ([142]).

2.7 Notebook: Other Clustering Algorithms

[1]:
```
from sklearn.datasets import load_iris, make_circles
from sklearn.metrics import adjusted_mutual_info_score
from sklearn.cluster import SpectralClustering
from sklearn.cluster import AffinityPropagation
from pyclustering.cluster.somsc import somsc
import  numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')

iris = load_iris()
```

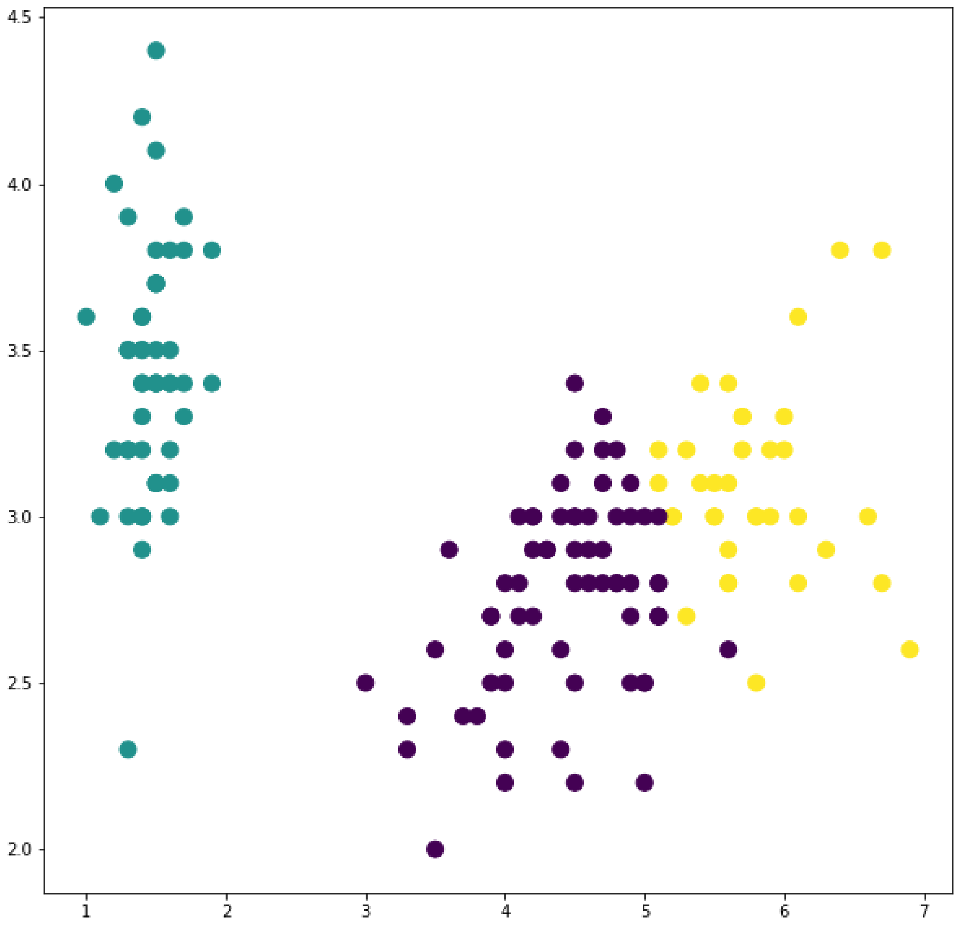
Spectral Clustering

Spectral clustering uses the distance matrix to define the Laplacian Matrix and transform the dataset by eigendecomposition. The implementation of scikit-learn uses K-means for obtaining the clusters n the transformed dataset (also a 'discretization' method can be applied).

[2]:
```
spec = SpectralClustering(n_clusters=3)
%time {spec.fit(iris['data'])}
labels = spec.fit_predict(iris['data'])
```

```
print("AMI=", adjusted_mutual_info_score(iris['target'], labels));
plt.figure(figsize=(10,10))
plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=labels, s=100);
```

```
CPU times: user 266 ms, sys: 256 ms, total: 523 ms
Wall time: 375 ms
AMI= 0.7842528489695738
```

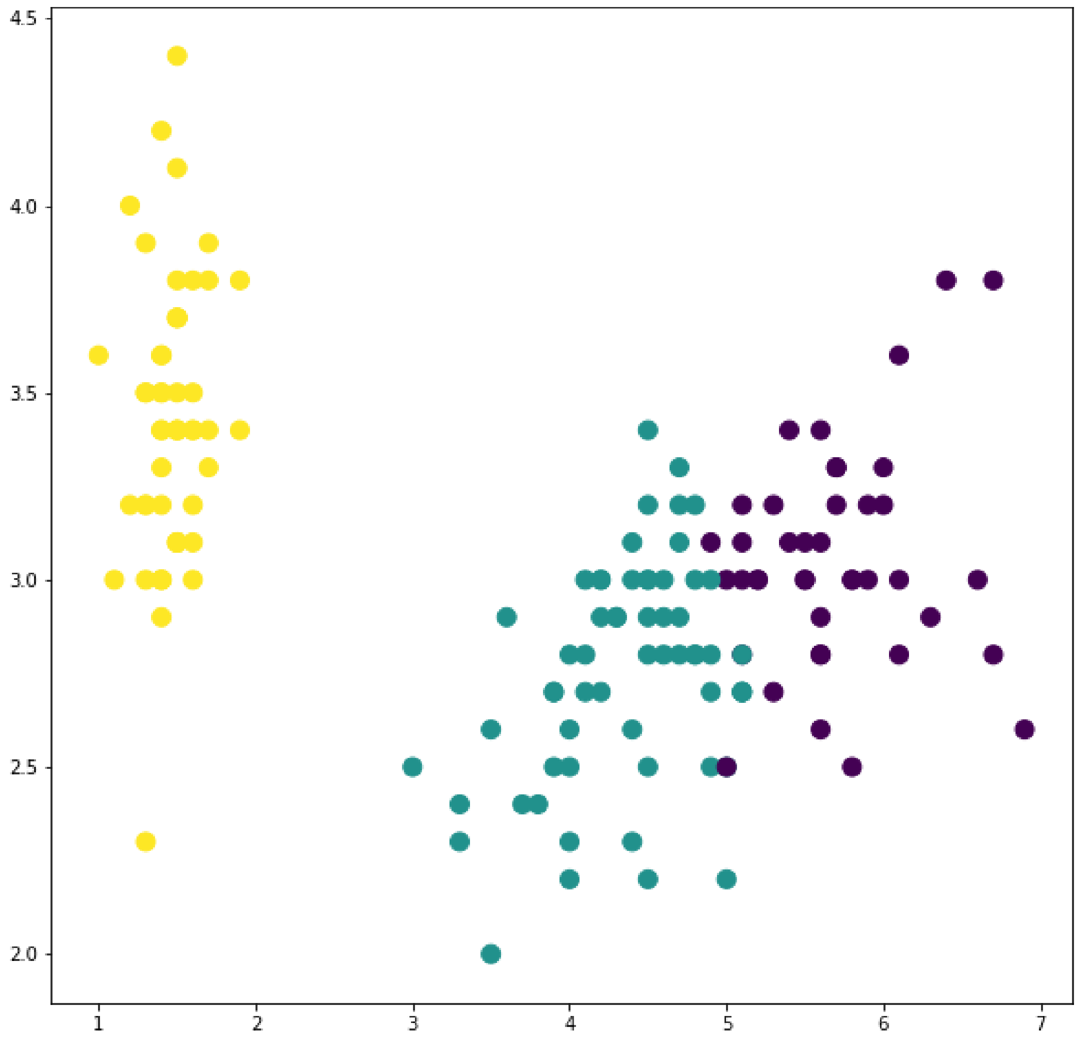


By default a RBF kernel is used to define the similarity matrix, but other possibilities like using the distances for only a number of neighbors or applying other kernel functions are possible. This makes this method related to Kernel K-means.

```
[3]: nn=30

spec = SpectralClustering(n_clusters=3, affinity='nearest_neighbors',␣
 ↪n_neighbors=nn)
%time {spec.fit(iris['data'])}

labels = spec.fit_predict(iris['data'])
print(adjusted_mutual_info_score(iris['target'], labels))
plt.figure(figsize=(10,10))
plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=labels, s=100);
```

```
CPU times: user 41.7 ms, sys: 238 µs, total: 42 ms
```

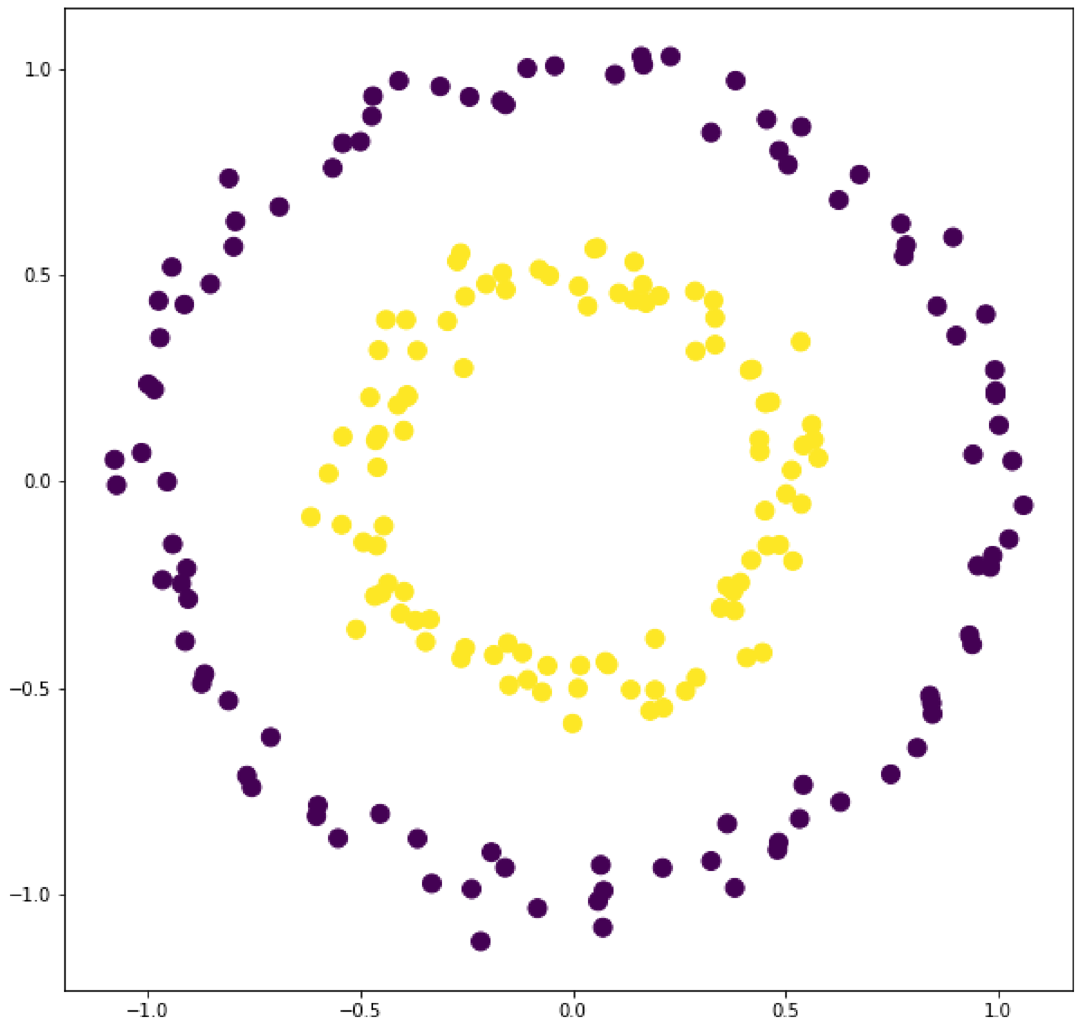
```
Wall time: 223 ms
0.7765248491347204
```



Depending on the transformation and parameters, the method is able to find non linearly separable clusters, but it is not always the case. You can see what happens with higher number of neighbors.

[4]:
```
circles, clabels = make_circles(n_samples=200, factor=.5, noise=.05)

nn=10

spec = SpectralClustering(n_clusters=2, affinity="nearest_neighbors",␣
 ↪n_neighbors=10)
labels = spec.fit_predict(circles)
print("AMI=", adjusted_mutual_info_score(clabels, labels))
plt.figure(figsize=(10,10))
plt.scatter(circles[:, 0], circles[:, 1], c=labels, s=100);
```

```
AMI= 1.0
```

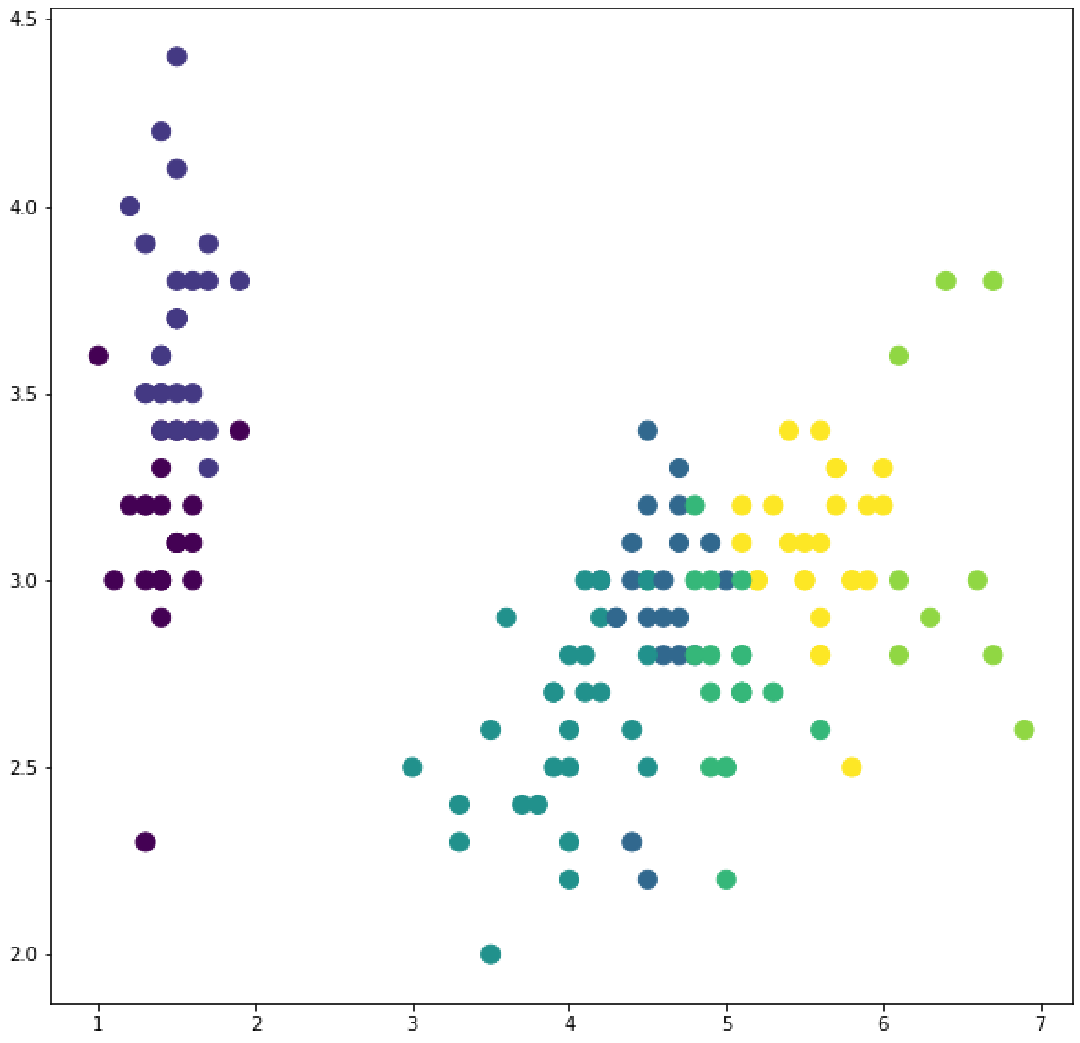


Affinity Propagation

Affinity propagation is related to probabilistical graphical models and is controled only by a parameter, the *damping factor* that controls how many clusters appear (the lower (0.5) the more clusters) buy the exact number of clusters obtained is determined by the algorithm. This example uses damping factor 0.5.

[5]:
```
aff= AffinityPropagation(damping=0.5)
%time {aff.fit(iris['data'])}

labels = aff.predict(iris['data'])
print ('Clusters=', len(np.unique(labels)))
print("AMI=", adjusted_mutual_info_score(iris['target'], labels))
plt.figure(figsize=(10,10))
plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=labels, s=100);
```

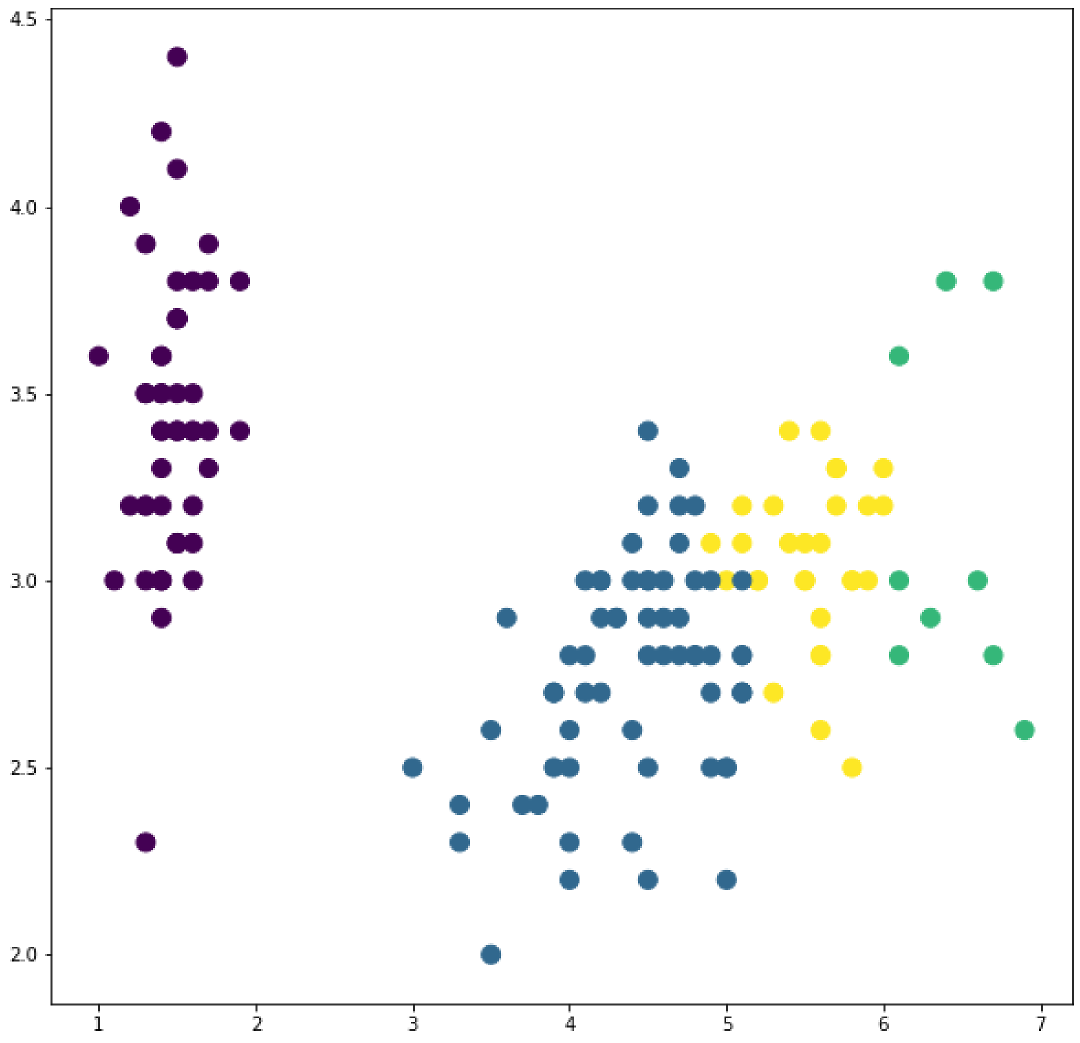
```
CPU times: user 41.2 ms, sys: 94.4 ms, total: 136 ms
Wall time: 70.2 ms
Clusters= 7
AMI= 0.5171731623283142
```



And now with damping factor 0.98 (a value of 1 does not returns clusters)

[6]:
```
aff= AffinityPropagation(damping=0.98)
%time {aff.fit(iris['data'])}
labels = aff.predict(iris['data'])
print ('Clusters=', len(np.unique(labels)))
print("AMI=",adjusted_mutual_info_score(iris['target'], labels))
plt.figure(figsize=(10,10))
plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=labels, s=100);
```

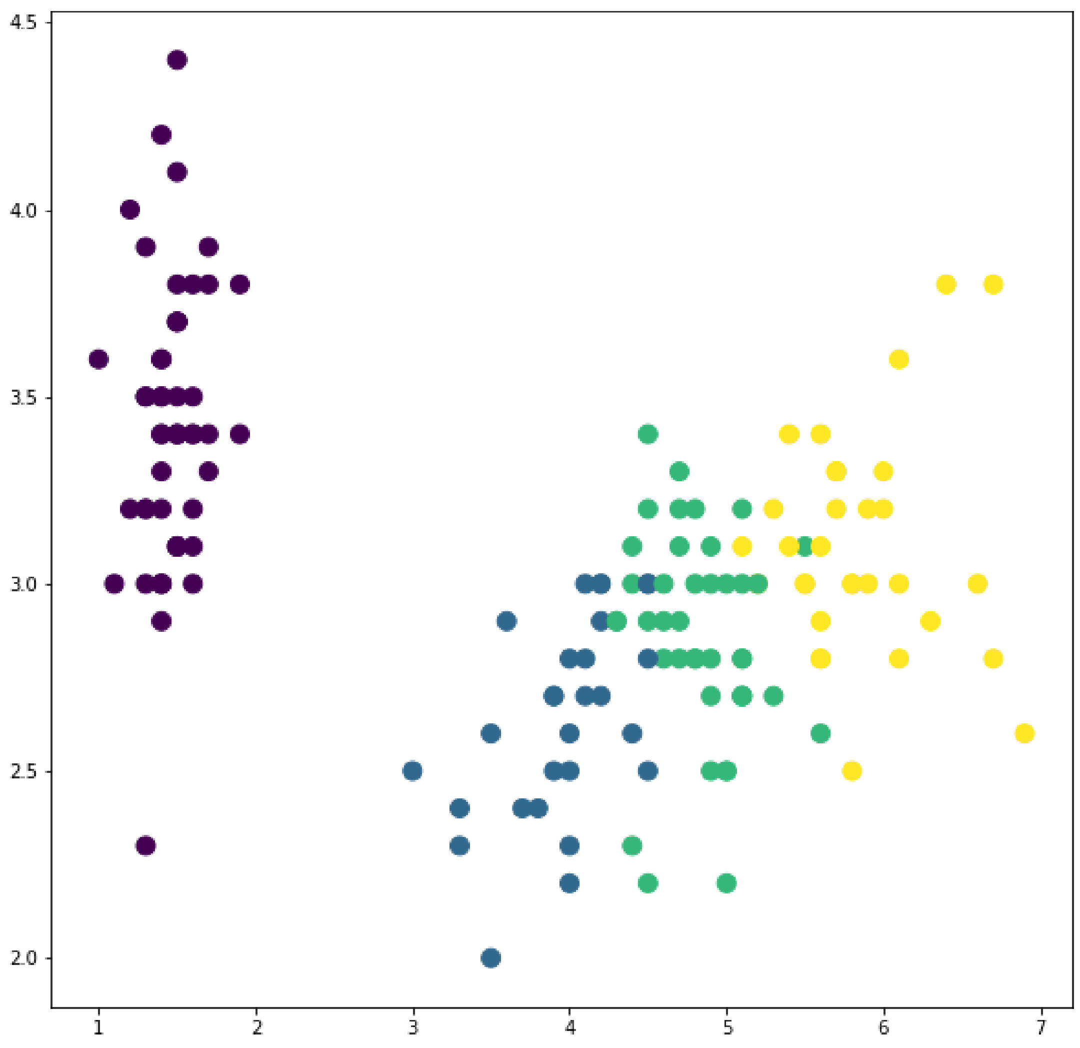
```
CPU times: user 160 ms, sys: 101 ms, total: 261 ms
Wall time: 125 ms
Clusters= 4
AMI= 0.6753193581954449
```



Self organizing maps

```
[7]: somsc_instance = somsc(iris['data'], 4, epouch=100);
     %time somsc_instance.process();
     clusters = somsc_instance.get_clusters()
     labels =  [l for _, l in sorted([(c, i) for i, cl in enumerate(clusters) for␣
      ↪c in cl])]
     print("AMI=",adjusted_mutual_info_score(iris['target'], labels))
     plt.figure(figsize=(10,10))
     plt.scatter(iris['data'][:, 2], iris['data'][:, 1], c=labels, s=100);
```

```
CPU times: user 9.5 ms, sys: 192 µs, total: 9.69 ms
Wall time: 153 ms
AMI= 0.6435272326577927
```



2.8 Application: Twitter Geoprofiling

This is an application to the study of the behavior of people living and visiting a large city. The goal is to analyze if there are patterns on their geographical behavior. The dataset used consist on the geolocalization of Tweets and Instagram posts inside a constrained geographical area. The data used is restricted to geographical information (latitude/longitude) and the time stamp of the post.

The process of analysis is divided in two steps, first the geographical discretization of the data for obtaining a representation of the users characteristics, and second, the discovery of different behavior profiles in the data.

The dataset is composed of a few millions of events during several months (represents less than 5% of the actual events) inside an area of 30 $\times$ 30 Km^2 of the city of Barcelona. Each event contains a geographical position and a time stamp. For the purpose of the analysis (geoprofiles) the actual data is difficult to analyze because the resolution of the coordinates is too fine. There is a low probability of having two events in the exact same place. Clustering geographically the events can help to make more sense of the data.

Being the data geographically distributed, there are few alternatives (clusters are of arbitrary shapes). The more adequate methods seem to be Density based clustering or Grid based clustering. The main problem is that the size of the dataset ($\sim$ 2.5 million events) could arise scalability issues, so a preliminary coarse grained clustering could be helpful (for example using

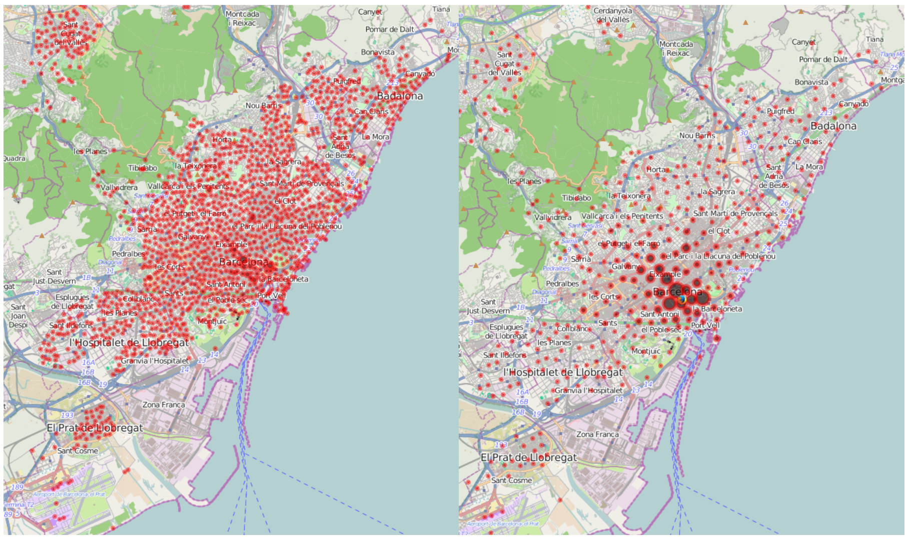


Figure 2.24: Clustering of events using the leader algorithm with different radius

k-means, or the leader algorithm).

The *epsilon* parameter of DBSCAN in this case has an intuitive meaning, how close geographically the points have to be, so they are considered related. The *minpoints* is more difficult, for this dataset, some areas have a lot of events and others have a few of them. Another curious thing about this dataset is that people generate events from almost everywhere, so we can end having just one cluster that connects everything.

Unfortunately the results using DBSCAN are not very good, apart from taking a long time for computing the results ($\sim$ 4 hours), only a few clusters are found, and many of examples are discarded as noise. As a consequence, it was decided that it was more informative to have a coarse discretization based on the leader algorithm. The granularity is defined by the radius parameter of the algorithm. The linear computational cost made possible to experiment with this parameter in order to find an adequate value. Figure 2.24 shows two examples of the resulting clusters.

We want to find groups of users that have similar behavior (in geographical position and time) along the period of data collection. The clusters obtained from the discretization can be the basis for the geographical profiles. Additionally, a discretization of time can provide a finer grained representation. Different representation can be used to generate the dataset:

- Presence/absence of a user in a place at a specific time interval
- Absolute number of visits of a user to a place in a time interval
- Normalized frequency of the visits of a user to a place in a time interval
- ...

Different choices of representation and discretization allow for different analysis. We have to remind here that there is not a unique way to partition a dataset, so having several choices allows analyzing the data from different perspectives.

For obtaining the profiles, it is more difficult to choose what clustering algorithm is adequate because there is not an intuition about how are distributed. We can explore different alternatives and analyze the results. Some choices will depend on:

- If the dataset generated is continuous or discrete

- The size of the dataset
- Our assumptions about the model that represents our goals (Shape of the clusters, Separability of the clusters/Distribution of examples)
- Interpretability/Representability of the clusters

Experts on the domain have to validate the results, but some general assumptions can be used to evaluate the alternatives. We will explore two possibilities:

- K-means algorithm (simple and efficient, spherical clusters)
- Affinity propagation clustering (adaptive and non predefined shape)

Clustering results depend on the chosen representation, our assumption is that several profiles should arise from the data, and the number of people for each profile should be large.

For K-means, if a large value of k is used, very small clusters are bound to appear. We can experiment with a reasonable range of values. From the results, most of the clusterings yield a very large cluster (most of the data) and several very small ones. The clustering that is closer to our assumptions is the one that uses a binary representation (presence/absence of a user in a region) normalized using TF-IDF. This obtains a relatively large number of clusters (around 25) with a large cluster, some medium-sized clusters and a few small ones. Profiles seem more biased to define people that moves around large regions, and the small clusters are not supported by many examples. Figure 2.25 shows some examples of the profiles obtained using this algorithm that present very localized behavior at different places of the city.

Using affinity propagation the only parameter of the algorithm to tune is the damping factor. We explored the range of possible values (from 0.5 to 1) to see how many clusters appear natural to this algorithm. Like for K-means the results vary depending on the representation used for the examples. There is a slightly larger number of clusters, and the examples are more evenly distributed among them. As for the previous algorithm, the clusterings closer to our assumptions are the ones obtained using the binary representation for the attributes. A larger number of clusters accounts for general and also more specific behaviors. They have a more evenly distributed of sizes, and the different profiles are supported by a number of examples enough to be significant. Figure 2.26 shows some examples of the profiles obtained using this algorithm.

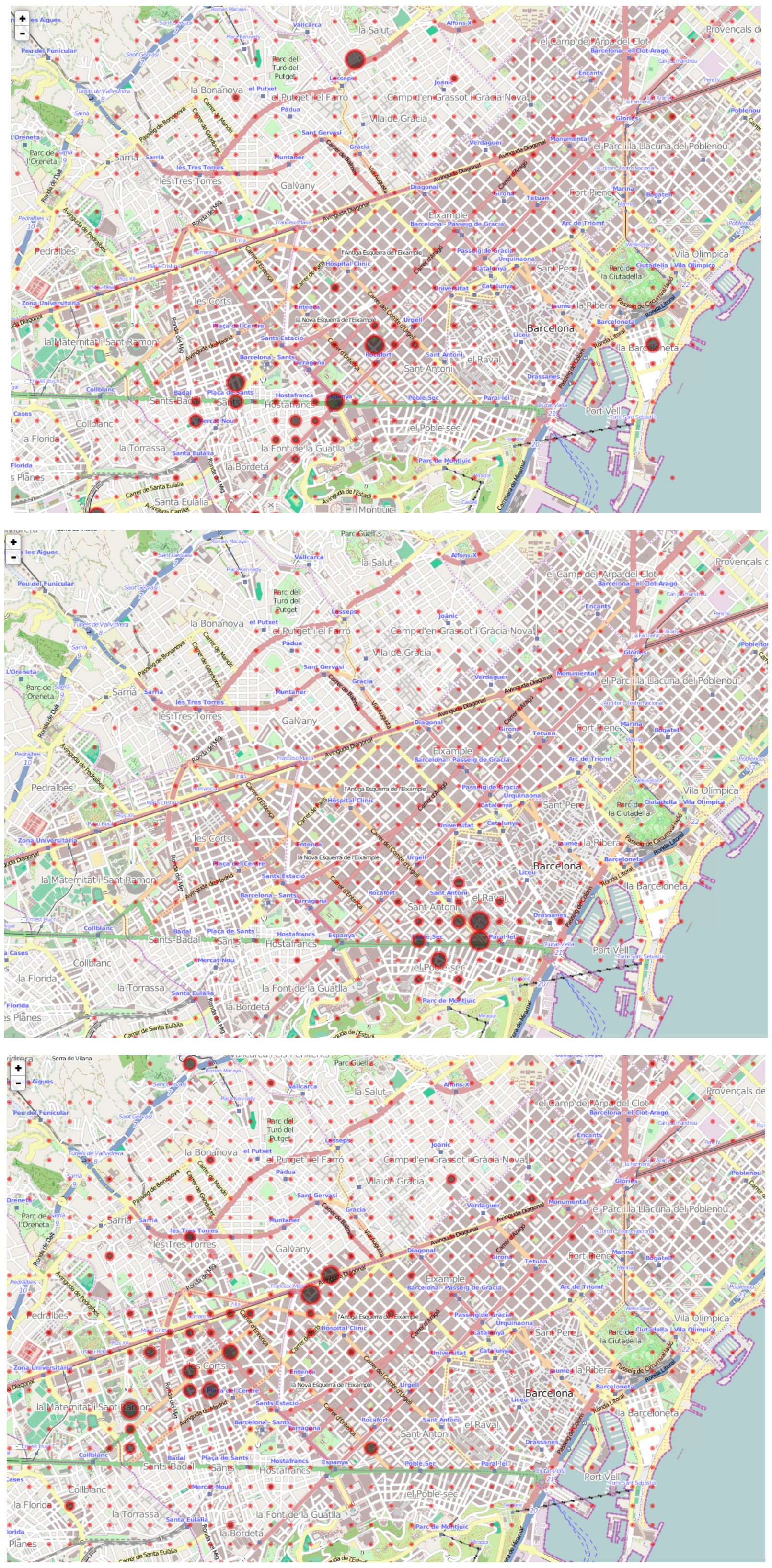


Figure 2.25: Examples of the geographical cluster obtained using K-means, they show different localized activity profiles.

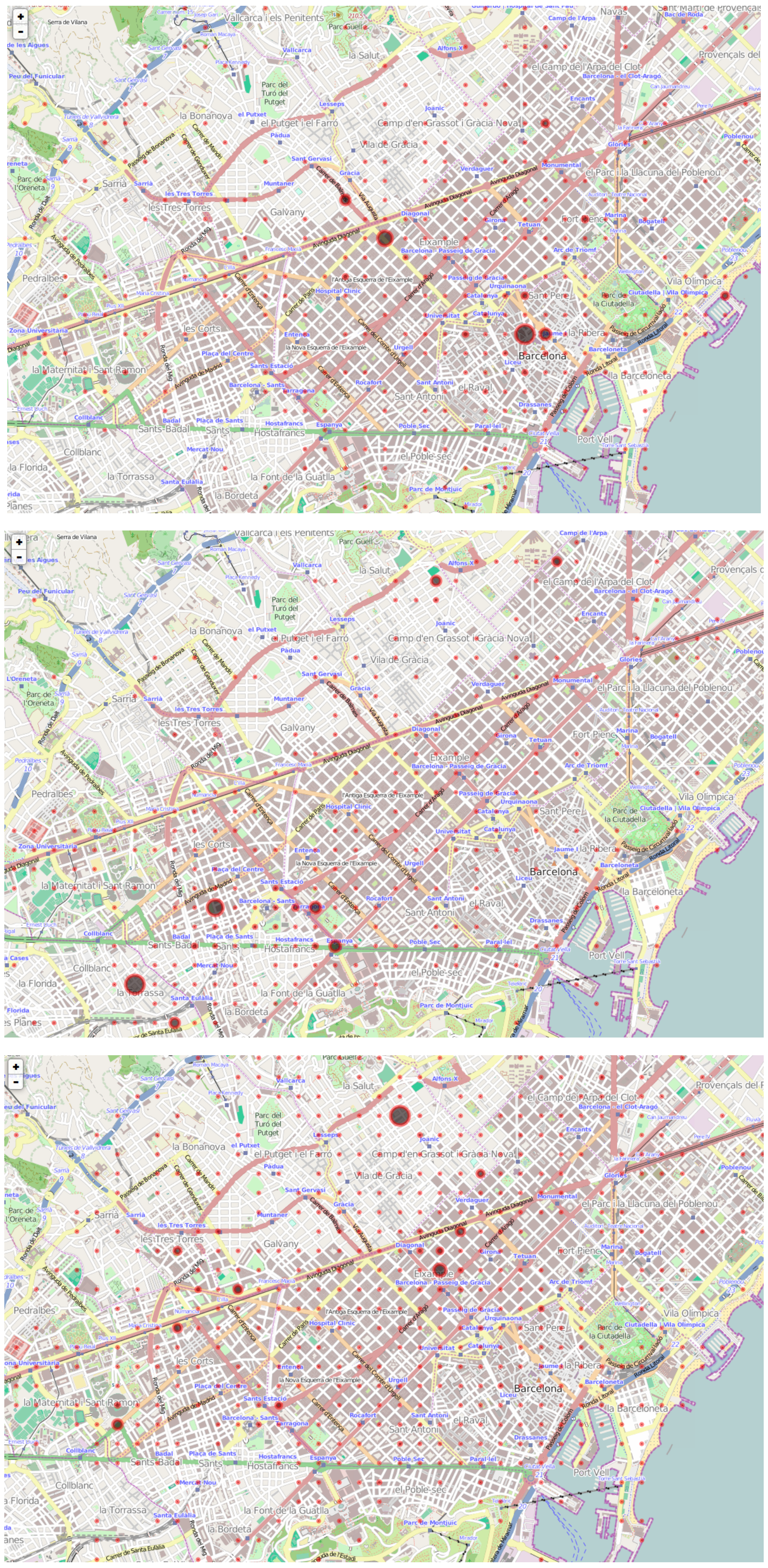
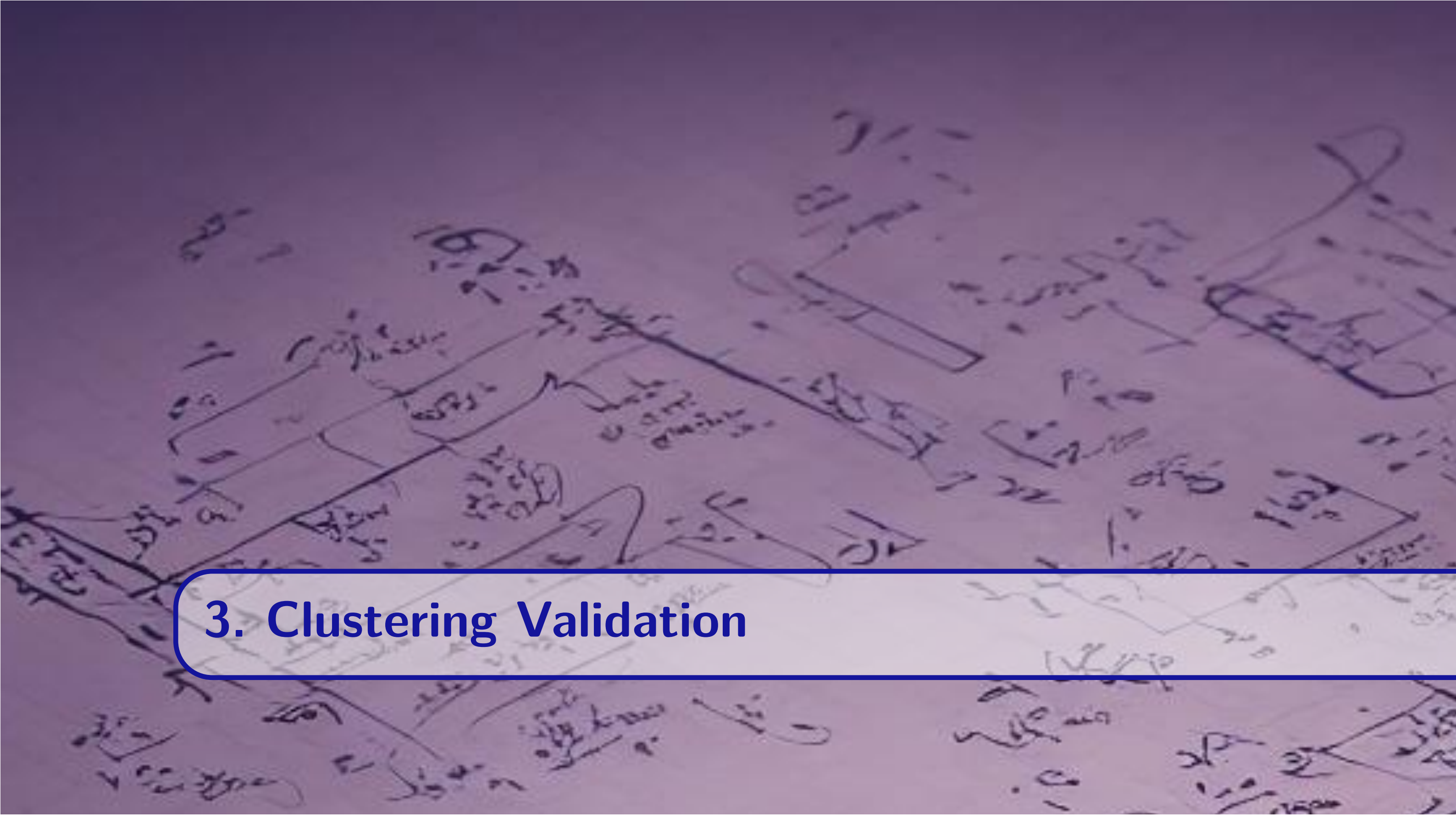


Figure 2.26: Examples of the geographical cluster obtained using affinity propagation, they show other more specific behavior profiles.



3. Clustering Validation

3.1 Assessing cluster quality

After processing a dataset using an unsupervised algorithm, the first question that arises is whether if an actual structure has been found on the data, or it is only a product of the bias of the algorithm. A disadvantage of the clustering task is that its evaluation is difficult. Supervised algorithms have available the actual labels, so they have an answer they can use to compare their predictions. This is not the case for unsupervised algorithms, there is no predefined model to compare with, the true result is unknown and moreover, it may depend on the context, or the actual task to perform with the discovered model.

Obviously, assessing the quality of the patterns discovered is important, so we need measures able to quantify cluster quality. There is not only one measure available for this task because of the lack of definition of the actual goal of cluster validation. We can measure how cluster-like are the patterns discovered, considering that the goal is to obtain well separated and compact clusters, but there is not a unique way to do this.

We can enumerate different applications for these measures. First, to avoid finding patterns in noise. This is something that will not be detected by most of the algorithms that we described in the previous chapter. For instance, if we apply K-means to a dataset we will always find a partition. We can suspect that something is wrong if we run the algorithm several times, and we obtain very different partitions, but because of the way the algorithm works, it will always result in a partition, even from random data. The same happens for hierarchical clustering, we can also suspect if when cutting the dendrogram it results in very strange partitions, but a dendrogram can always be computed.

The second reason for clustering validation measures is to compare different clustering algorithms, or the results obtained by the tuning of the parameters of a single algorithm. Each algorithm has its own bias, and their parameters can have a huge impact on the discovered patterns, so it is interesting to know if is there is a difference among the structure of the data that different choices are able to capture. These measures are also useful when the labels for the data are available. We can measure how well the assumed structure of the data is discovered by the chosen algorithm, but the same criteria used for supervised learning can be also used in this case instead of unsupervised measures.

These measures can also be used for model selection (for instance, to determine the number of clusters). All of them capture heuristically some characteristic of the patterns that can be optimized. We can use this optimization as a means of finding the best parameters for an algorithm considering that the measure really captures the kind of patterns we are interested in.

3.2 Measures for assessing clusterness

Before clustering a dataset we can test if there are actually clusters to be found. We have to test the hypothesis of the existence of patterns in the data versus a dataset that is uniformly distributed (homogeneous distribution). This can be measured by the *Hopkins Statistic*. The procedure to compute this measures is as follows:

1. Sample n points (p_i) from the dataset (D) uniformly and compute the distance to their nearest neighbor ($d(p_i)$)
2. Generate n points (q_i) uniformly distributed in the space of the dataset and compute their distance to their nearest neighbors in D ($d(q_i)$)
3. Compute the quotient:

$$H = \frac{\sum_{i=1}^{n} d(p_i)}{\sum_{i=1}^{n} d(p_i) + \sum_{i=1}^{n} d(q_i)} \tag{3.1}$$

If the data is uniformly distributed, the value of H will be around 0.5. For instance, figure 3.1 shows the value for the Hopkins statistic for a dataset of five clusters with different variance, so they increasingly overlap. The value of the statistic decreases as data are more mixed, but it still shows clustering tendency compared with uniformly generated data as can be seen on the rightmost plot of the figure.

3.3 Measures for clustering validation

We can divide the different criterion to evaluate the quality of a clustering in three groups:

- Quality measures based on the characteristics of the examples (Internal criteria)
- Methods that compare with a model partition/labeled data (External criteria)
- Measures to compare clusterings with each other (Relative criteria)

3.3.1 Internal validity criteria

These measures use the properties that are expected in a good clustering: compact and well separated groups. This is the bias that one way or another all clustering algorithm use. The indices compute a heuristic function that is optimized by a good clustering.

The computation of the criteria is based on the characteristics of the model of the groups and the statistical properties of the attributes of the data. These measures usually assume that we have continuous attributes, and a distance defined for the data, so the statistical distribution of the values of the attributes and the distribution of the examples/clusters distances can be obtained.

Some of these indices correspond directly to the objective function optimized by the clustering algorithm, for instance:

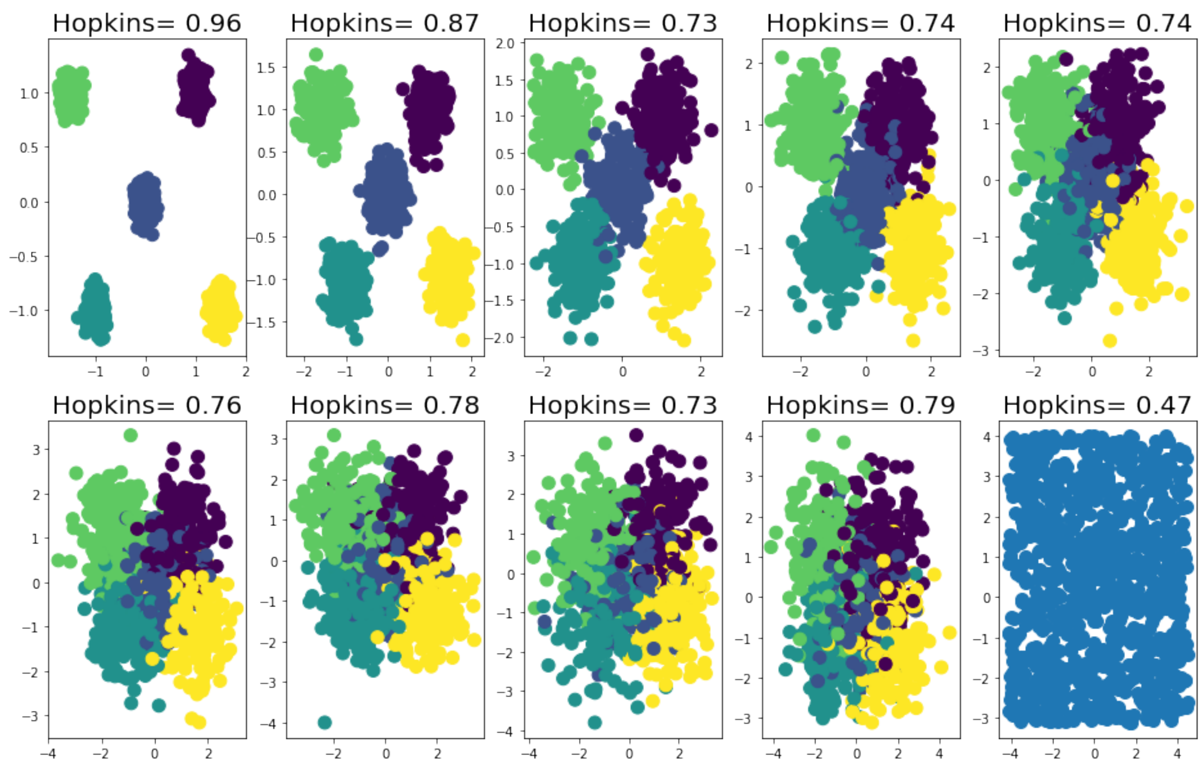


Figure 3.1: Value of the Hopkins statistic for a dataset with 5 clusters with different variance and data with uniform distribution (last plot)

- Quadratic error/Distortion (k-means)

$$SSE = \sum_{k=1}^{k} \sum_{\forall x_i \in C_k} \parallel x_i - \mu_k \parallel^2 \tag{3.2}$$

- Log likelihood (Mixture of gaussians/EM)

For prototype based algorithms several measures can be obtained using the statistical distribution of the values of the cluster prototypes. They are based on three quantities defined by the scatter matrices of the clusters: interclass distance (within distances), intraclass distance (between distances) and cross-intercluster distance.

$$S_{W_k} = \sum_{\forall x_i \in C_k} (x_i - \mu_k)(x_i - \mu_k)^T \tag{3.3}$$

$$S_{B_k} = |C_k|(\mu_k - \mu)(\mu_k - \mu)^T \tag{3.4}$$

$$S_{M_{k,l}} = \sum_{\forall i \in C_k} \sum_{\forall j \in C_l} (x_i - x_j)(x_i - x_j)^T \tag{3.5}$$

The simplest criteria uses the trace of the interclass distance (S_W) and the intraclass distance (S_B) (trace criteria):

$$Tr(S_W) = \frac{1}{K} \sum_{i=1}^{K} S_{W_k} \tag{3.6}$$

$$Tr(S_B) = \frac{1}{K} \sum_{i=1}^{K} S_{B_k} \tag{3.7}$$

A cluster is better if it has a lower overall intracluster distance, and a higher overall intercluster distance. This means compact and separated clusters.

From the cross-intercluster distance it is defined the determinant criteria

$$Det(S_M) = \frac{1}{K^2}\sum_{i=1}^{K}\sum_{j=1}^{K} S_{M_{i,j}} \tag{3.8}$$

A higher value means that the examples from each cluster are separated from the examples of the other clusters.

These criteria have their limitations, basically they fail when natural clusters are not well separated or have different densities. Also, it is not possible to compare clusters that have a different number of partitions, for example intraclass distance decreases monotonically with the number of partitions, and is zero (the lower the better) when there are as many partitions as examples.

More complex criteria that try to avoid these problems have been defined, these are the more frequently used:

- Calinski Harabasz index (interclass/intraclass distance ratio)

  $$CH = \frac{\sum_{i=0}^{K} |C_i| \times \|\mu_i - \mu\|^2/(K-1)}{\sum_{k=1}^{K}\sum_{i=0}^{|C_i|} \|x_i - \mu_i\|^2/(N-K)} \tag{3.9}$$

  This index computes a ratio between the interclass and intraclass distances and takes in account the number of clusters in the partition.

- Davies-Bouldin criteria (maximum interclass/cross-intraclass distance ratio)

  $$\bar{R} = \frac{1}{K}\sum_{i=1}^{K} R_i \tag{3.10}$$

  where

  $$R_{ij} = \frac{S_{W_i} + S_{W_j}}{S_{M_{ij}}} \tag{3.11}$$

  $$R_i = \max_{j:j\neq i} R_{ij} \tag{3.12}$$

  This index takes into account only the maximum ration of the separation among each pair of the clusters respect to their compactness.

- Silhouette index (maximum class spread/variance)

  $$S = \frac{1}{N}\sum_{i=0}^{N} \frac{b_i - a_i}{max(a_i, b_i)} \tag{3.13}$$

  Where

  $$a_i = \frac{1}{|C_j| - 1} \sum_{y \in C_j, y \neq x_i} \|y - x_i\| \tag{3.14}$$

  $$b_i = \min_{l \in H, l \neq j} \frac{1}{|C_l|} \sum_{y \in C_l} \|y - x_i\| \tag{3.15}$$

with $x_i \in C_j, H = \{h : 1 \leq h \leq K\}$

This index considers how the examples are scattered in the clusters, so the more compact the better. Its value can be plotted also for each individual example, so we can observe how the examples fit the clusters. In figure 3.2 we can observe this value for a dataset of three well separated clusters using two, three and four clusters. In the plot for two clusters we can see how the mean score is lower, and a cluster has merged two of the clusters, it can be seen how different is the individual score for the two groups of examples. In the plot for four clusters the mean score is even lower, and a cluster has been split in two, affecting the variance of the score of the examples.

In the literature, they can be found more than 30 different indices (new ones appear every year) and there is a consistent effort on studying their performance. A recent study ([5]) has exhaustively studied many of these indices, finding that some have a performance significantly better that others and some show a similar performance (not statistically different). The study concludes that Silhouette, Davies-Bouldin and Kalinski-Harabasz perform well in a wide range of situations.

Example

Figure 3.3 shows the plot of the trace of the Within Scatter (SW) matrix, the Calinski-Harabasz (CH) and Davis-Bouldin (DB) indices for a dataset with 5 well separated clusters. When clusters are well separated, all the criteria work well. Usually the criteria for deciding what clusters are better is to choose the optimal value (minimum or maximum depending on the direction of the criteria) or detecting a jump in the plot (this is usually called the *elbow* method). Figure 3.4 shows the same indices for non well separated clusters, for the SW criteria now the change in the tendency of the values is not so clear as before, the CH criteria now has a jump in the correct number of clusters instead of a minimum, but the DB criteria shows correctly the number of clusters.

3.3.2 External criteria

They measure if a clustering is similar to a model partition P. This mean that either we have the labels corresponding to the examples, or we want to assess the difference among different ways of obtaining a partition of the data.

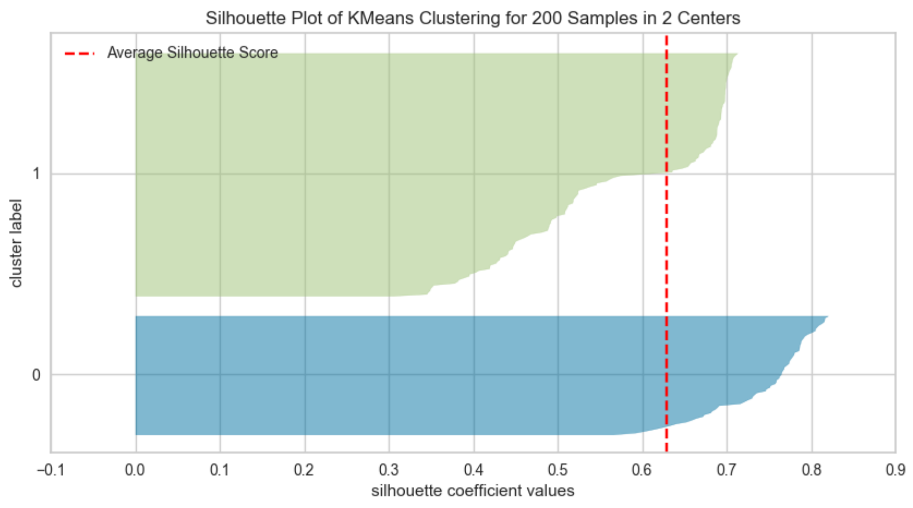
This second possibility is useful for model assessment, comparing the results of using different parameters of the same algorithm or completely different algorithms. For instance, it can be used to assess the sensitivity to initialization. These indices are independent of the attributes representing the data and how the cluster model is described, this means that it can be used to compare any clustering algorithm and dataset.

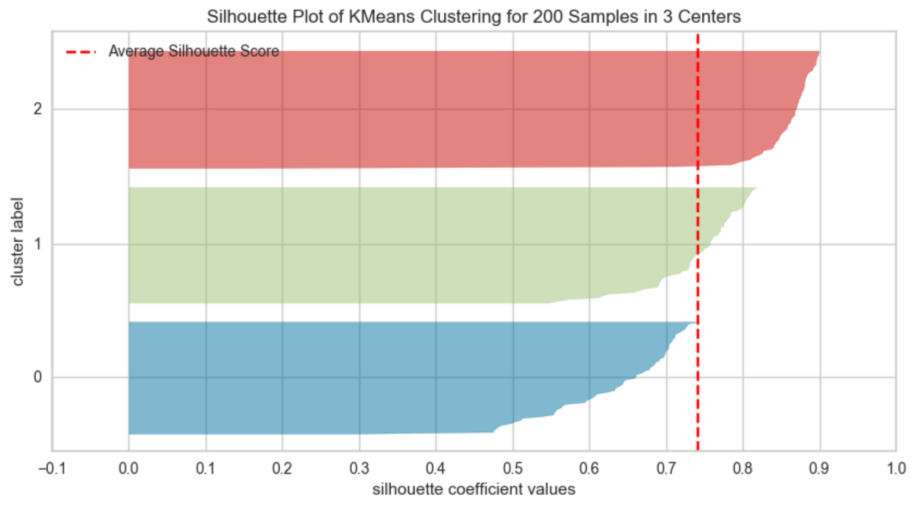
Pairs matching

All the indices are based on the coincidence of each pair of examples in the groups of two clusterings. The computations are based on four values:

- The two examples belong to the same class in both partitions (a)
- The two examples belong to the same class in C, but not in P (b)
- The two examples belong to the same class in P, but not in C (c)
- The two examples belong to different classes in both partitions (d)

From these quantities different measures can be defined, these are the most used:





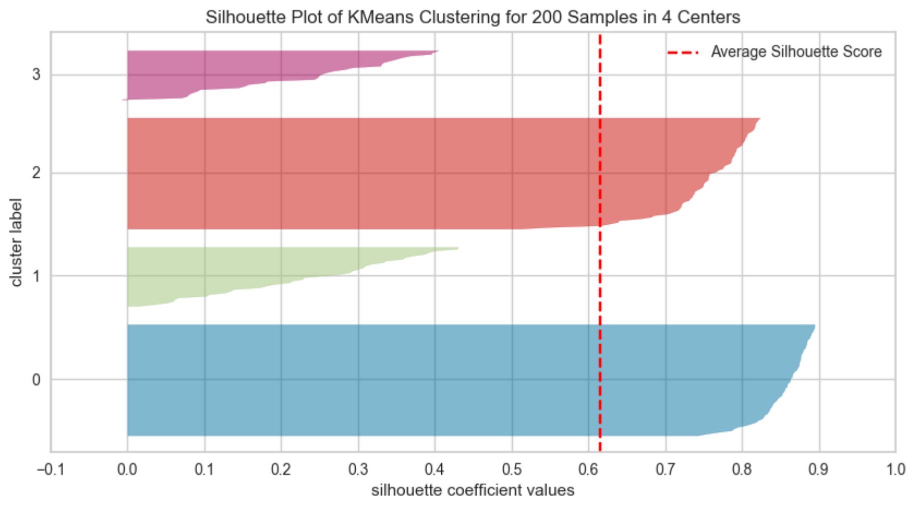


Figure 3.2: Silhouette index for individual examples for a dataset with three well separated clusters using two, three and four clusters

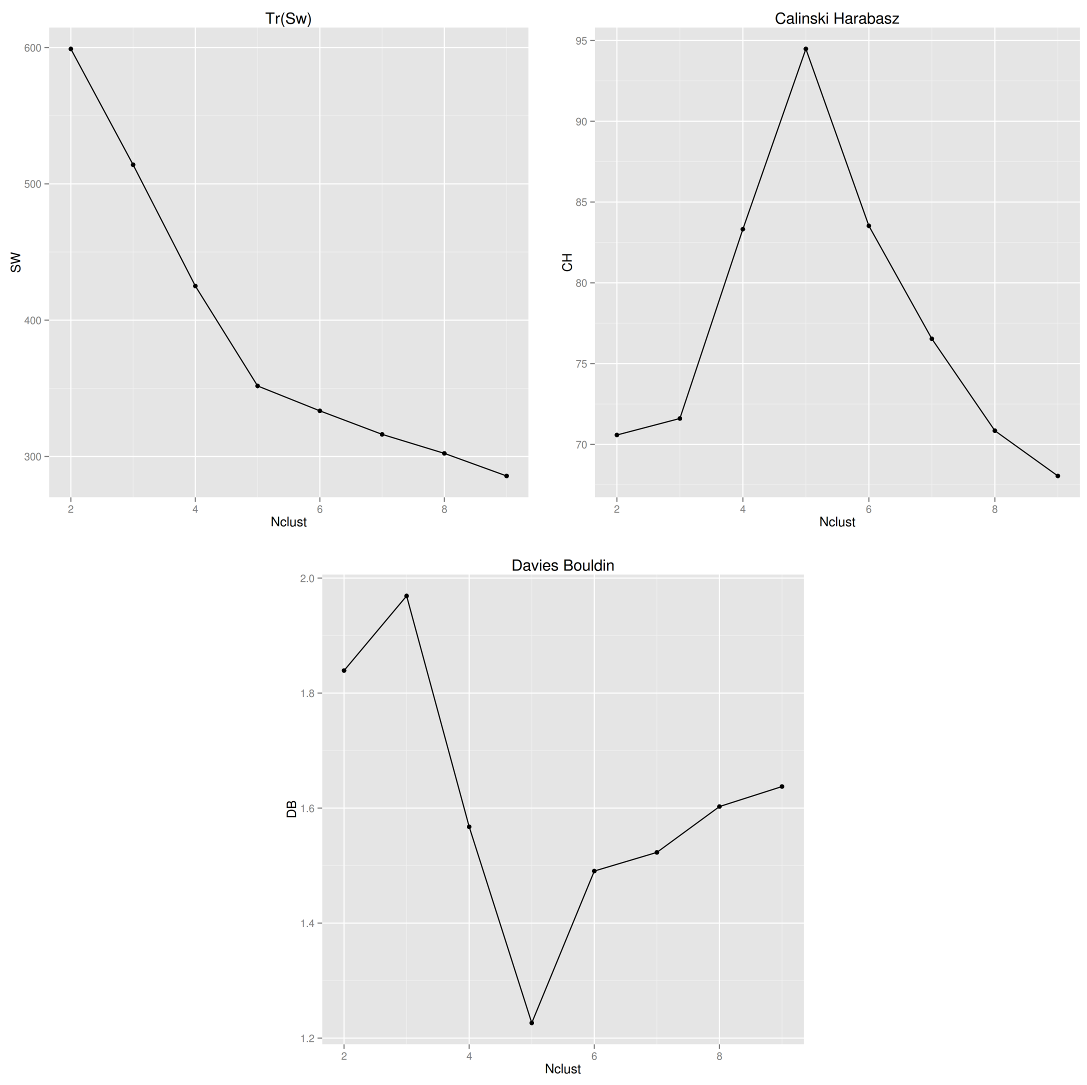


Figure 3.3: Trace of the Within Scatter matrix, Calinski-Harabasz and Davis-Bouldin indices for a dataset with 5 well separated clusters

- Rand/Adjusted Rand statistic:

$$R = \frac{(a+d)}{(a+b+c+d)} \quad ARand = \frac{a - \frac{(a+c)(a+b)}{a+b+c+d}}{\frac{(a+c)+(a+b)}{2} - \frac{(a+b)(a+c)}{a+b+c+d}} \tag{3.16}$$

  The adjusted version takes into account that the agreement of two random partitions is not zero and scales the range of values of the index correspondingly.

- Jaccard Coefficient:

$$J = \frac{a}{(a+b+c)} \tag{3.17}$$

  This index assumes that only are important the events when two examples agree on both partitions or completely disagree. Usually the events when two examples belong to different clusters in both partitions are larger than the rest, overshadowing the contribution of the other possibilities.

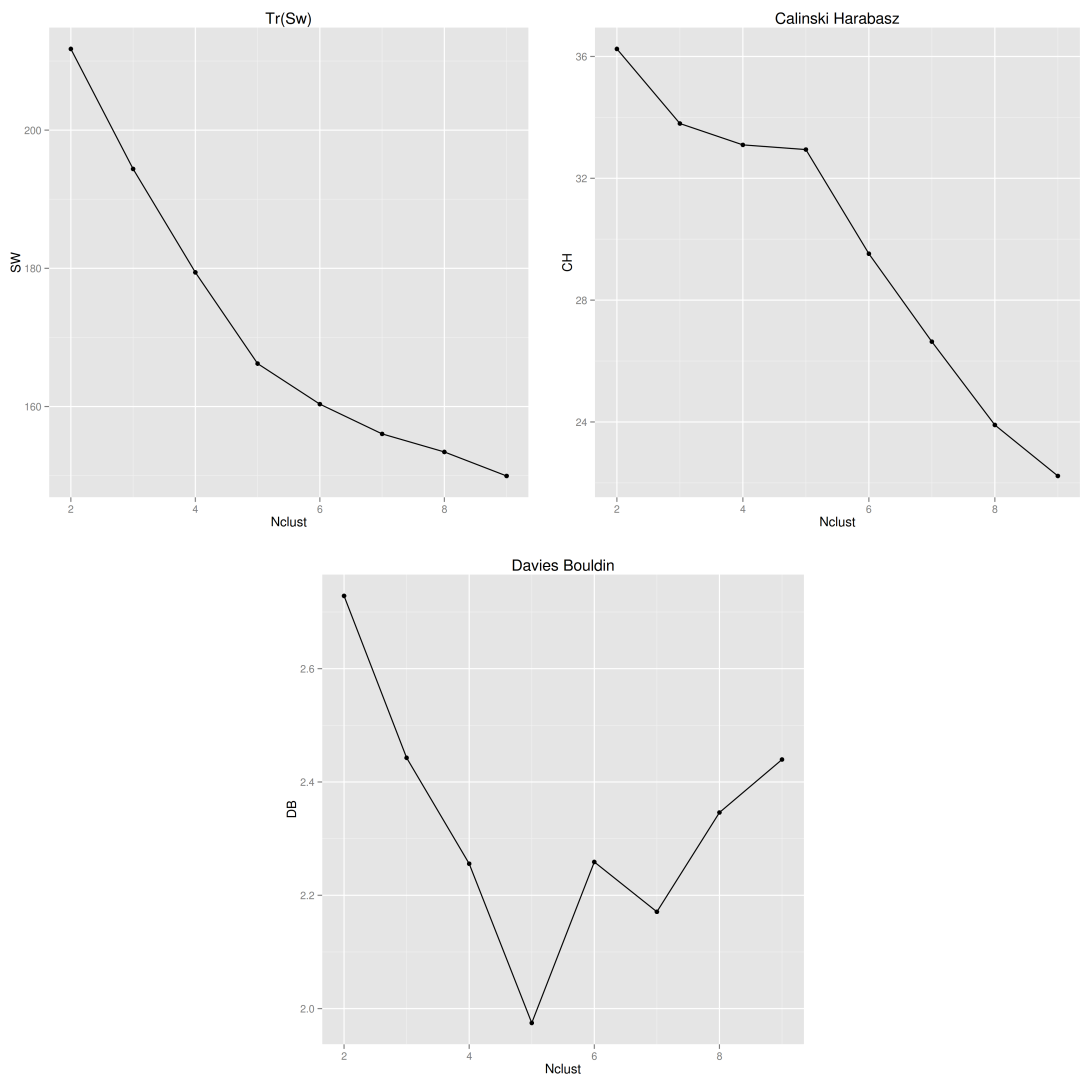


Figure 3.4: Trace of the Within Scatter matrix, Calinski-Harabasz and Davis-Bouldin indices for a dataset with 5 noisy non well separated clusters

- Folkes and Mallow index:

$$FM = \sqrt{\frac{a}{a+b} \cdot \frac{a}{a+c}} \tag{3.18}$$

  This index uses the geometric mean instead of the arithmetic.

Information Theory

An alternative way to measure the agreement of partitions is to use information theory measures. Assuming that the partition correspond to discrete statistical distributions, their similarity from this perspective can be assimilated to agreement. We can define the Mutual Information between two partitions as:

$$MI(Y_i, Y_k) = \sum_{X_c^i \in Y_i} \sum_{X_{c'}^k \in Y_k} \frac{|X_c^i \cap X_{c'}^k|}{N} \log_2\left(\frac{N|X_c^i \cap X_{c'}^k|}{|X_c^i||X_{c'}^k|}\right) \tag{3.19}$$

and the Entropy of a partition as:

$$H(Y_i) = - \sum_{X_c^i \in Y_i} \frac{|X_c^i|}{N} \log_2(\frac{|X_c^i|}{N}) \tag{3.20}$$

where $X_c^i \cap X_{c'}^k$ is the number of objects that are in the intersection of the two groups. From these quantities different measures can be defined as:

- Normalized Mutual Information:

$$NMI(Y_i, Y_k) = \frac{MI(Y_i, Y_k)}{\sqrt{H(Y_i)H(Y_k)}}$$

- Variation of Information:

$$VI(C, C') = H(C) + H(C') - 2I(C, C')$$

- Adjusted Mutual Information:

$$AMI(U, V) = \frac{MI(U, V) - E(MI(U, V))}{\max(H(U), H(V)) - E(MI(U, V))}$$

Example

As an example, figure 3.5 shows the values of the Adjusted Rand Index (ARI) and the Adjusted Mutual Information (AMI) for a dataset with 5 clusters using different mixings of labels, 10% of randomly changed labels, 25% of randomly changed labels and a random labeling of the data. Both scores result in similar values decreasing rapidly as the mixing increases. As expected the random assignment of labels results in a score of 0 because both indexes are adjusted for randomness versions of the original measures.

3.3.3 Relative Criteria

These previous measures allow comparing partitions without using the actual description of the partitions. An alternative when we have continuous attributes and a centroid based description of partitions is to use the distances among the centroids of different partitions and their variance as a measure of agreement. In this case this can be interpreted as measure of the correlation among the partitions. The following measures can be used for this purpose:

- Modified Hubert Γ statistic

$$\Gamma = \frac{2}{N \cdot N - 1} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} P(i,j) \cdot Q(i,j) \tag{3.21}$$

  Where $P(i,j)$ is the distance matrix of the dataset and $Q(i,j)$ is a $N \times N$ matrix with the distance between the centroids the examples x_i and x_j belong to.

- Normalized Hubert Γ statistic

$$\Gamma = \frac{\frac{2}{N \cdot N - 1} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} (P(i,j) - \mu_P) \cdot (Q(i,j) - \mu_Q)}{\sigma_P \sigma_Q} \tag{3.22}$$

  A significant increase of this value indicates the number of clusters

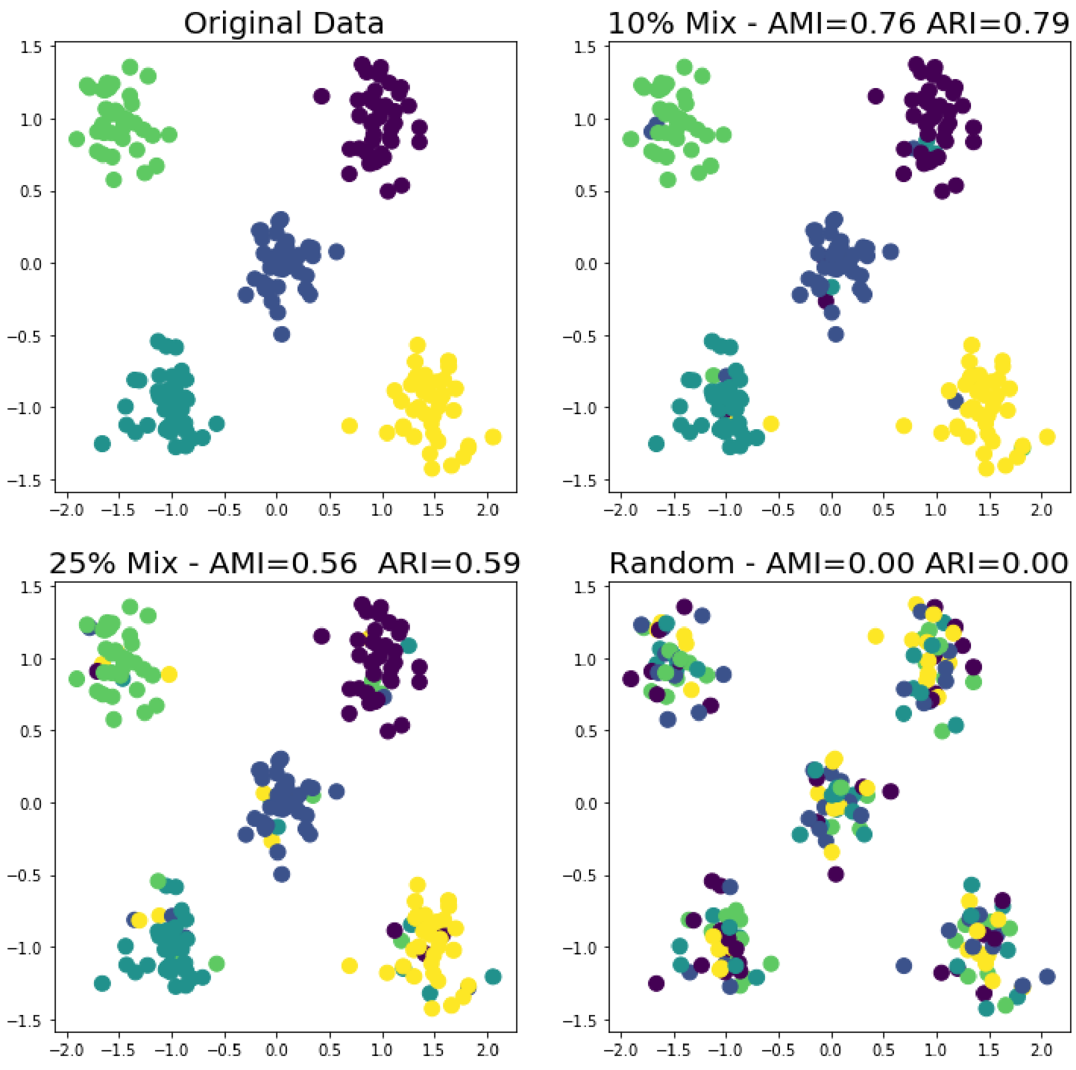


Figure 3.5: Adjusted Rand Index and Adjusted Mutual Information for different mixings of labels (10%, 25% and random) clusters

- *SD* index

$$SD(n_c) = Dis(n_{max}) \cdot Scat(n_c) + Dis(n_c) \tag{3.23}$$

Where n_{max} is the maximum number of clusters, $\sigma(v_i)$ is the variance of a cluster, $\sigma(X)$ is the variance of the data set, D_{max} and D_{min} are the maximum and minimum cluster distances and

$$Scat(n_c) = \frac{\frac{1}{n_c}\sum_{i=1}^{n_c} \|\sigma(v_i)\|}{\|\sigma(X)\|} \tag{3.24}$$

$$Dis(n_c) = \frac{D_{max}}{D_{min}} \sum_{i=1}^{n_c} \left(\sum_{j=1}^{n_c} \|v_i - v_j\| \right)^{-1} \tag{3.25}$$

3.4 Assessing the number of clusters

3.4.1 Classical measures

A topic related to cluster validation is to decide if the number of clusters obtained is the correct one. This is part of model assessment/validation, and it is important specially for the algorithms that have this value as a parameter. The usual procedure is to compare the characteristics of clusterings with different number of clusters. Internal criteria indices are preferred in this comparison. A plot of the variation of these indices with the number of partitions can reveal how the quality of the clustering changes. The optimal value in the plot according to a specific criterion can reveal what number of clusters is more probable for the dataset.

Several of the internal validity indices explained in section 3.3.1 can be used for this purpose as shown in figures 3.3 and 3.4, for instance the Calinsky Harabasz index or the Silhouette index. The usual way to apply these indices is to look for a jump in the value of the criteria (commonly known as a *knee* or *elbow*) or to detect the number of clusters that yields the optimal value (minimum or maximum).

There are specialized criteria for this purpose that use the within class scatter matrix (S_W) such as:

- Hartigan index:

$$H(k) = \left[\frac{S_W(k)}{S_W(k+1)} - 1 \right] (n-k-1) \tag{3.26}$$

  This criterion is frequently used in hierarchical clustering to decide the level where to cut the dendrogram to obtain a flat partition.

- Krzanowski Lai index:

$$KL(k) = \left| \frac{DIFF(k)}{DIFF(k+1)} \right| \tag{3.27}$$

  with $DIFF(k) = (k-1)^{2/p} S_W(k-1) - k^{2/p} S_W(k)$

These are relative indices that compare the change in quality when the number of partitions is increased in one unit. The optimal number of clusters is decided also looking for a significant jump in the plot of its values. The effectiveness of these indices depends on how separable are the clusters and sometimes the plot presents different optimal number of clusters.

3.4.2 The Gap statistic

An alternative to assess the number of clusters in a dataset is to define it as a hypothesis test. One way is to compare the clustering with the expected distribution of data given the null hypothesis (no clusters). The procedure needs the generation of different clusterings of the data increasing the number of clusters and to compare each one to clusterings of datasets (B) generated with a uniform distribution. It is assumed that there will be a difference for the measures taken from the original data and the uniform distributed data and this difference will be significantly larger for the correct number of clusters.

A very successful test that uses this method is the Gap statistic ([127]), it measures the difference on the sum of inter class distances (S_W), and is defined as:

$$Gap(k) = (1/B) \sum_b log(S_W(k)_b) - log(S_W(k)) \tag{3.28}$$

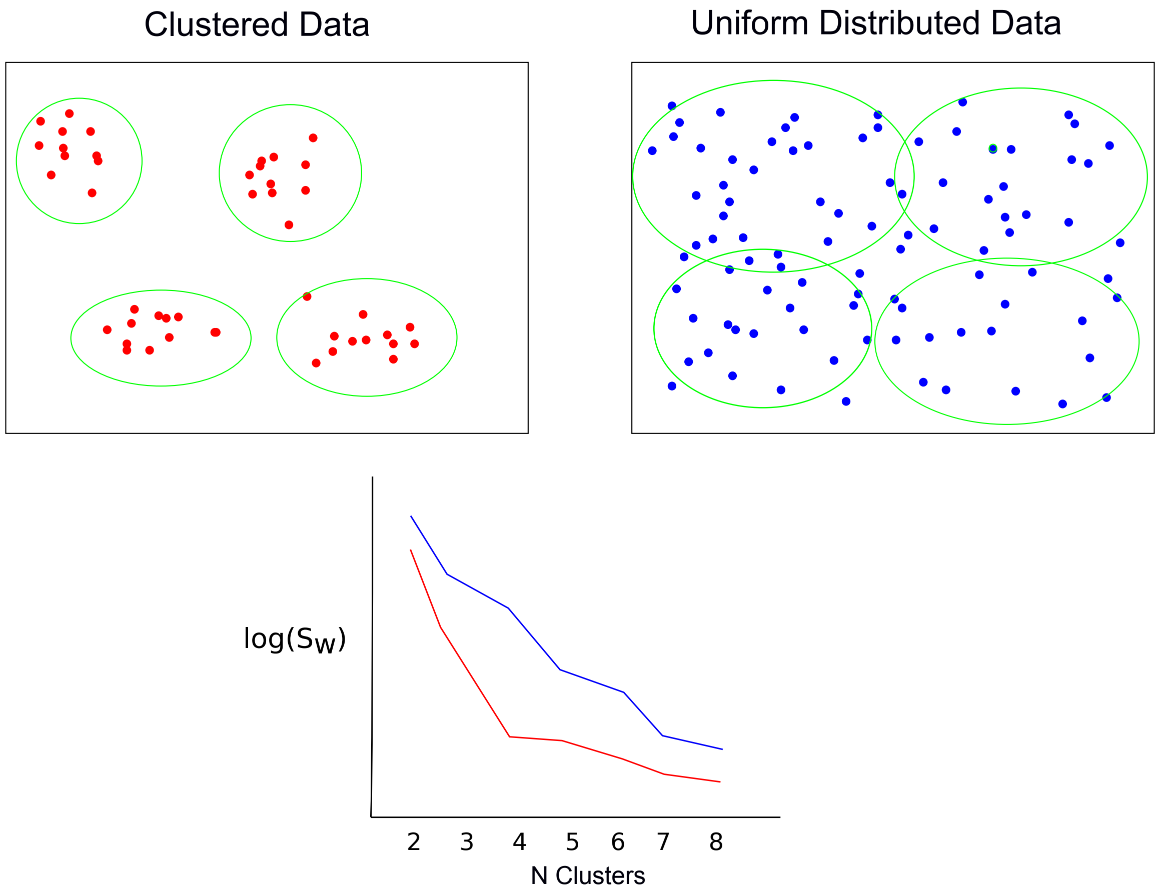


Figure 3.6: The gap statistic measures the diference between the interclass distances for clusters from the original data and for clusters from uniform distributed data, four clusters for this example.

We are averaging the measure of all the generated samples (B) and comparing with the original data. The standard deviation (sd_k) of $\sum_b log(S_W(k)_b)$ is defined s_k as:

$$s_k = sd_k\sqrt{1+1/B} \tag{3.29}$$

The probable number of clusters is the smallest number that holds:

$$Gap(k) \geq Gap(k+1) - s_{k+1} \tag{3.30}$$

Figure 3.6 shows a representation of the procedure, in this case we obtain four clusters for the data and all the samples. Plotting the measure will show where the largest difference appears.

3.4.3 Cluster stability

A third method is to use the concept of *cluster stability* ([84, 97]). The idea behind this method is that if the model chosen for clustering a dataset is correct, it should be stable for different samplings of the data. The procedure is to obtain different subsamples of the data, cluster them and test how stable they are.

There are two methods:

- Using disjoint samples:

The dataset is divided in two disjoint samples that are clustered separately. Several indices can be defined to assess stability, for instance, to the distribution of the number of neighbors that belong to the complementary sample. An adequate number of clusters should generate similar partitions, so there should not be a large deviation in the number of neighbors that belong to one or the other sample.

- Using non-disjoint samples:

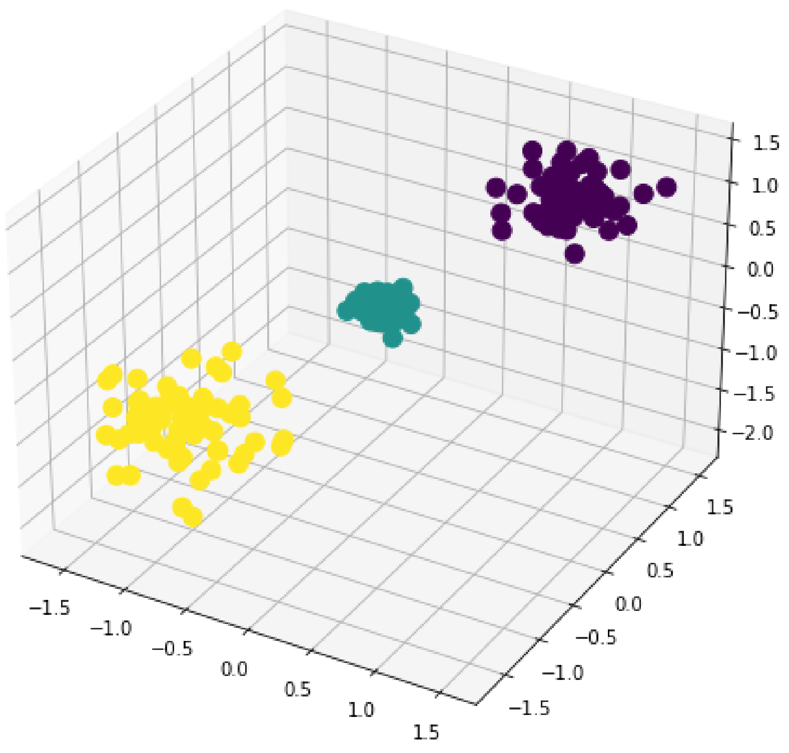
  The dataset is divided in three disjoint samples (S_1,S_2, S_3). Two clusterings are obtained from $S_1 \cup S_3$, $S_2 \cup S_3$. Different indices can be defined about the coincidence of the common examples in both partitions. The adequate number of clusters should have many common examples among the different samples.

3.5 Notebook: Clustering Validation

[2]:
```python
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from mpl_toolkits.mplot3d import Axes3D
from kemlglearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from sklearn.metrics import adjusted_mutual_info_score, silhouette_score
from sklearn.metrics import calinski_harabasz_score,davies_bouldin_score
from sklearn.metrics import adjusted_rand_score, normalized_mutual_info_score
from kemlglearn.metrics import  scatter_matrices_scores
from sklearn.datasets import make_moons
from sklearn.manifold import Isomap

import numpy as np
import warnings
warnings.filterwarnings('ignore')
```

We will play with data blobs with different characteristics

[3]:
```python
blobs, blabels = make_blobs(n_samples=200, n_features=3,␣
 ↪centers=[[1,1,1],[0,0,0],[-1,-1,-1]], cluster_std=[0.2,0.1,0.3])
fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
plt.scatter(blobs[:, 0], blobs[:, 1], zs=blobs[:, 2], depthshade=False,␣
 ↪c=blabels, s=100);
```

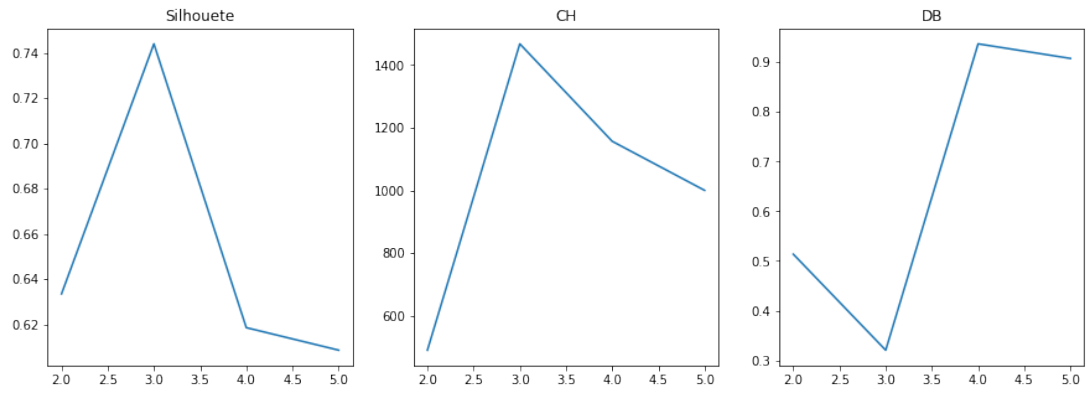


These are very well separated blobs very easy to discover using k-means

[4]:
```
km = KMeans(n_clusters=3, n_init=10, random_state=0)
labels = km.fit_predict(blobs)
print("AMI=", adjusted_mutual_info_score(blabels, labels))
```

```
AMI= 1.0000000000000002
```

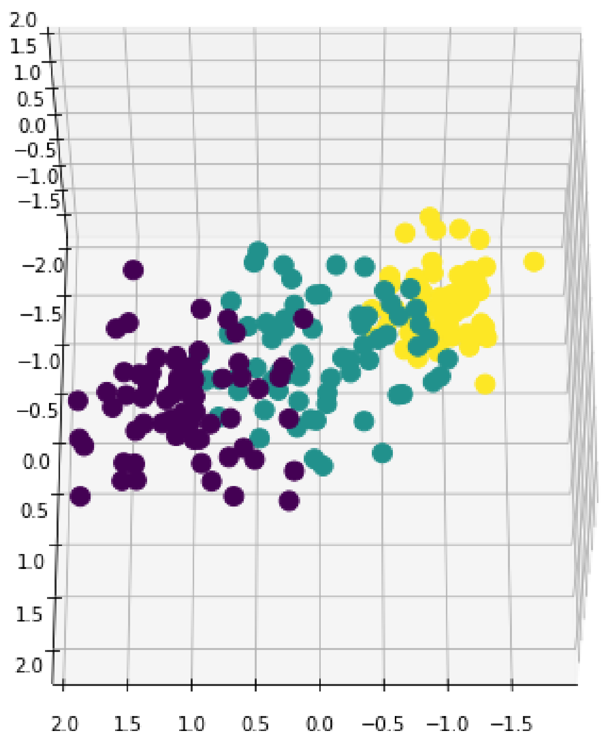
[5]:
```
lscores = []
nclusters = 5
indices = ['Silhouete', 'CH', 'DB']
for nc in range(2,nclusters+1):
    km = KMeans(n_clusters=nc, n_init=10, random_state=0)
    labels = km.fit_predict(blobs)
    lscores.append((
        silhouette_score(blobs, labels),
        calinski_harabasz_score(blobs, labels),
        davies_bouldin_score(blobs, labels)))

fig = plt.figure(figsize=(15,5))
for i,ind in enumerate(indices):
    ax = fig.add_subplot(1,3,i+1)
    plt.title(ind)
    plt.plot(range(2,nclusters+1), [x[i] for x in lscores]);
```



The Silhouette (higher), CH (higher) and DB (lower) indices agree on that there are three clusters on the data

Let's introduce some overlapping among the clusters

```
[6]: blobs, blabels = make_blobs(n_samples=200, n_features=3,␣
      ↪centers=[[1,1,1],[0,0,0],[-1,-1,-1]], cluster_std=[0.4,0.45,0.3])
     fig = plt.figure(figsize=(8,8))
     ax = fig.add_subplot(111, projection='3d')
     ax.view_init(60, 90)
     plt.scatter(blobs[:, 0], blobs[:, 1], zs=blobs[:, 2], depthshade=False,␣
      ↪c=blabels, s=100);
```

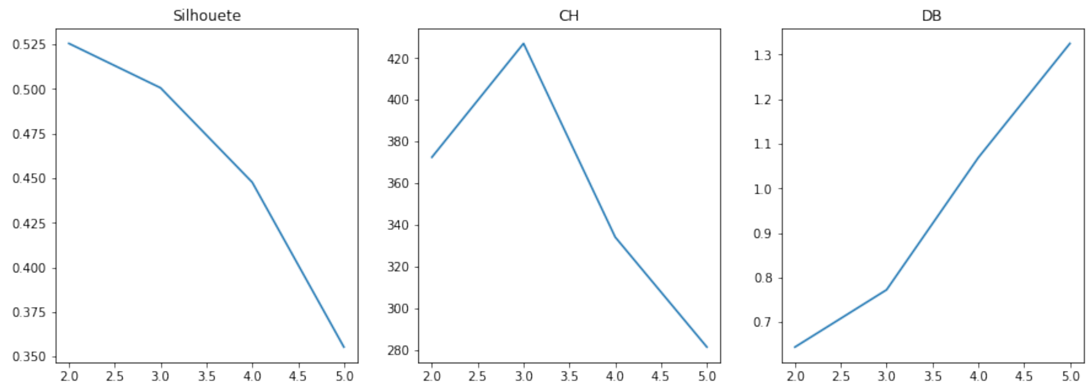


Now is a little bit harder for k-means to discover the clusters

[7]:
```
km = KMeans(n_clusters=3, n_init=10, random_state=0)
labels = km.fit_predict(blobs)

print("AMI=",adjusted_mutual_info_score(blabels, labels))
```

```
AMI= 0.8501107518561282
```

[8]:
```
lscores = []
nclusters = 5
indices = ['Silhouete', 'CH', 'DB']
for nc in range(2,nclusters+1):
    km = KMeans(n_clusters=nc, n_init=10, random_state=0)
    labels = km.fit_predict(blobs)
    lscores.append((
        silhouette_score(blobs, labels),
        calinski_harabasz_score(blobs, labels),
        davies_bouldin_score(blobs, labels)))

fig = plt.figure(figsize=(15,5))
for i,ind in enumerate(indices):
  ax = fig.add_subplot(1,3,i+1)
  plt.title(ind)
  plt.plot(range(2,nclusters+1), [x[i] for x in lscores]);
```

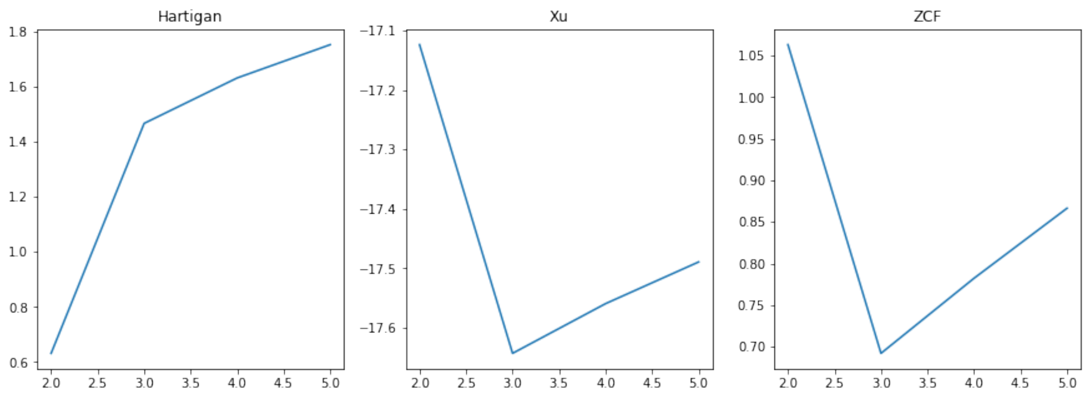


Now only CH indicex says that there is three clusters on the dataset, Silhouette and DB only see two.

These are the classical Hartigan index (used fequently for hierarchical clustering) and two other recent published indices

[9]:
```
lscores = []
nclusters = 5
indices = ['Hartigan', 'Xu', 'ZCF']
for nc in range(2,nclusters+1):
    km = KMeans(n_clusters=nc, n_init=10, random_state=0)
    labels = km.fit_predict(blobs)
    lscores.append(scatter_matrices_scores(blobs, labels, indices= indices))

fig = plt.figure(figsize=(15,5))
```

```
for i, ind in enumerate(indices):
  ax = fig.add_subplot(1,3,i+1)
  plt.title(ind)
  plt.plot(range(2,nclusters+1), [x[ind] for x in lscores]);
```

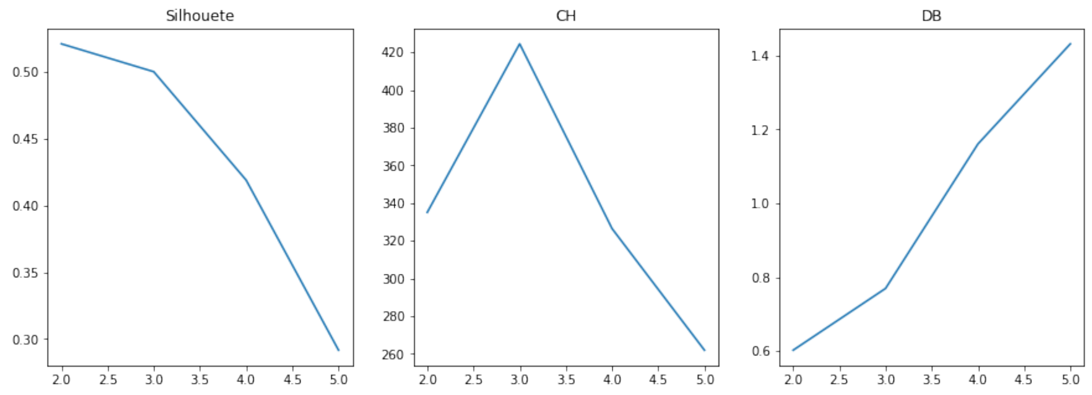


GMM does not have more luck discovering the noisy clusters.

```
[10]: gmm = GaussianMixture(n_components=3, covariance_type='diag', random_state=0)
gmm.fit(blobs)
print("AMI=", adjusted_mutual_info_score(blabels, labels))
```

```
AMI= 0.7438637496864413
```

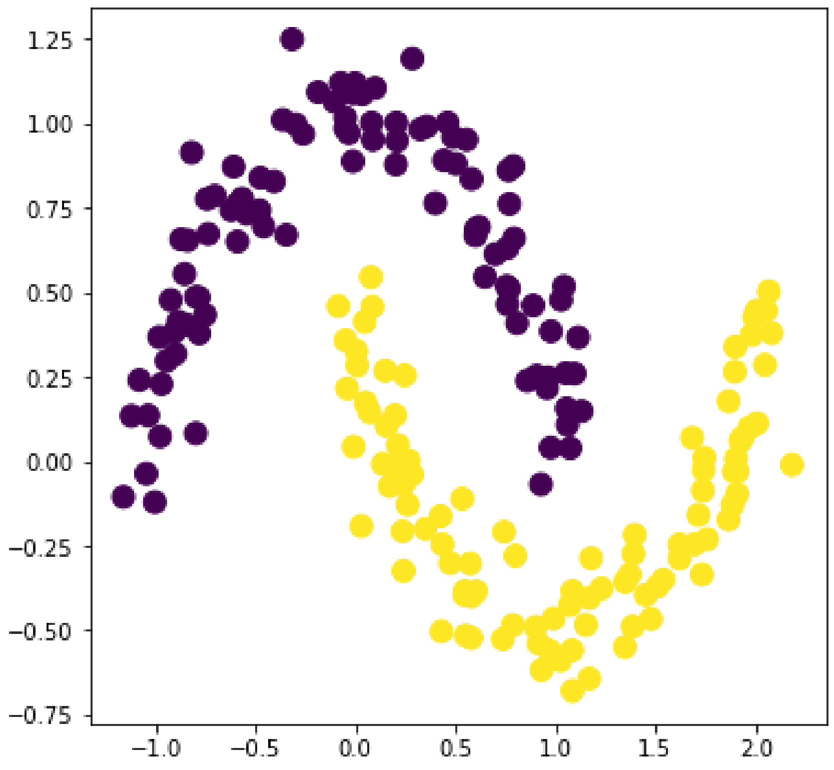
```
[11]: lscores = []
nclusters = 5
indices = ['Silhouete', 'CH', 'DB']
for nc in range(2,nclusters+1):
    gmm = GaussianMixture(n_components=nc, covariance_type='diag',␣
 ↪random_state=0)
    gmm.fit(blobs)
    labels = gmm.predict(blobs)
    lscores.append((
        silhouette_score(blobs, labels),
        calinski_harabasz_score(blobs, labels),
        davies_bouldin_score(blobs, labels)))

fig = plt.figure(figsize=(15,5))
for i,ind in enumerate(indices):
  ax = fig.add_subplot(1,3,i+1)
  plt.title(ind)
  plt.plot(range(2,nclusters+1), [x[i] for x in lscores]);
```



And the indices have similar results.

Now for non linear data.

[12]:
```
moons, mlabels = make_moons(n_samples=200, noise=0.1)
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111)
plt.scatter(moons[:, 0], moons[:, 1], c=mlabels, s=100);
```

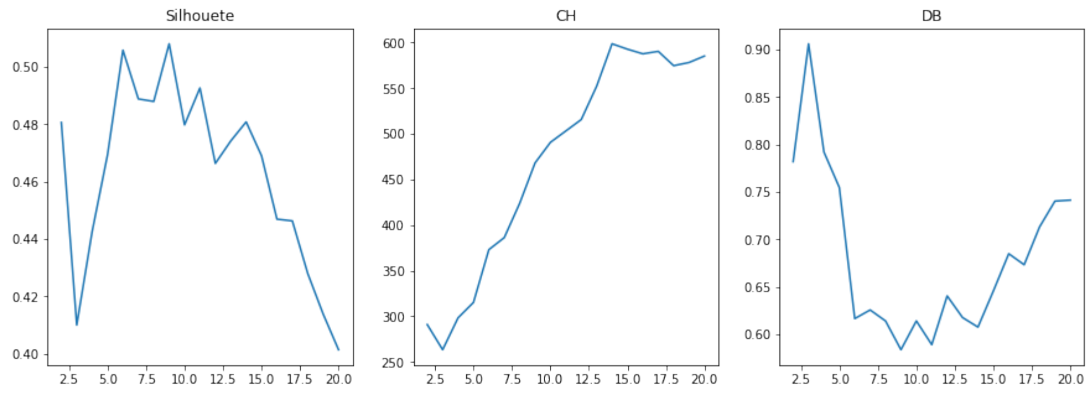


Obviously the clusters obtained by K-means have little to do with the actual labels

[13]:
```
km = KMeans(n_clusters=2, n_init=10, random_state=0)
labels = km.fit_predict(moons)

print("AMI=", adjusted_mutual_info_score(mlabels, labels))
```

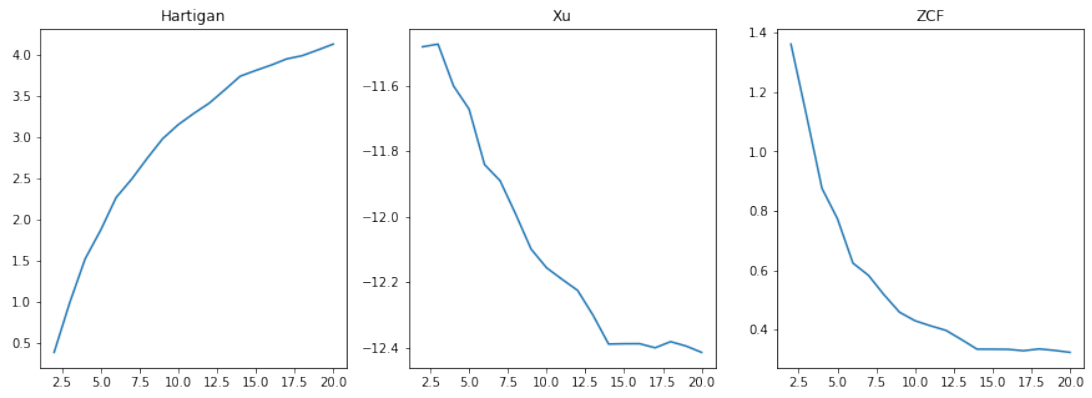
```
AMI= 0.14838067760419865
```

[14]:
```
lscores = []
nclusters = 20
indices = ['Silhouete', 'CH', 'DB']
for nc in range(2,nclusters+1):
    km = KMeans(n_clusters=nc, n_init=10, random_state=0)
    labels = km.fit_predict(moons)
    lscores.append((
        silhouette_score(moons, labels),
        calinski_harabasz_score(moons, labels),
        davies_bouldin_score(moons, labels)))

fig = plt.figure(figsize=(15,5))
for i,ind in enumerate(indices):
  ax = fig.add_subplot(1,3,i+1)
  plt.title(ind)
  plt.plot(range(2,nclusters+1), [x[i] for x in lscores]);
```



Now the scores result in a high number of clusters, Silhouette and DB are close to agreeing on (maybe) 8 clusters, for CH the more the better (12-13?)

[15]:
```
lscores = []
nclusters = 20
indices = ['Hartigan', 'Xu', 'ZCF']
for nc in range(2,nclusters+1):
    km = KMeans(n_clusters=nc, n_init=10, random_state=0)
    labels = km.fit_predict(moons)
    lscores.append(scatter_matrices_scores(moons, labels, indices=␣
 ↪['Hartigan', 'Xu', 'ZCF']))

fig = plt.figure(figsize=(15,5))
for i, ind in enumerate(indices):
  ax = fig.add_subplot(1,3,i+1)
  plt.title(ind)
  plt.plot(range(2,nclusters+1), [x[ind] for x in lscores]);
```



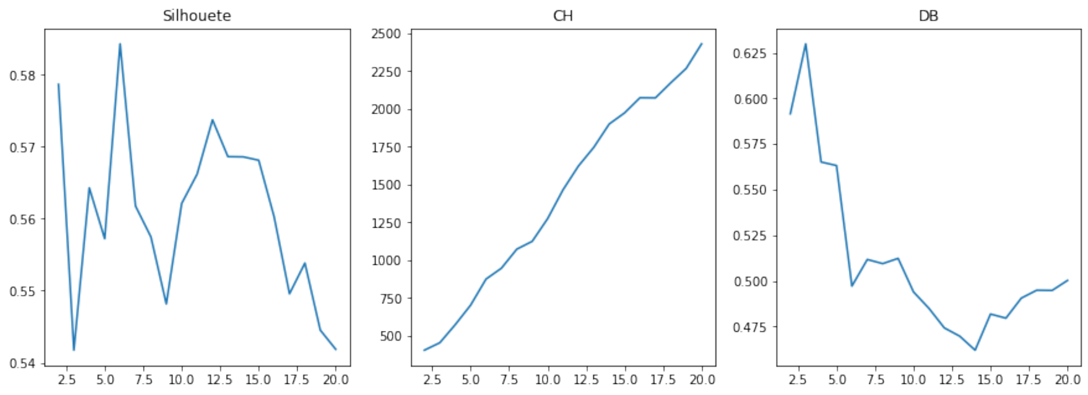
The other indices also agree on the more the better.

We can apply a non linear transformation to the data.

[16]:
```python
iso = Isomap(n_components=2, n_neighbors=7)
fdata = iso.fit_transform(moons)

km = KMeans(n_clusters=2, n_init=10, random_state=0)
labels = km.fit_predict(fdata)

print("AMI=", adjusted_mutual_info_score(mlabels, labels))
```

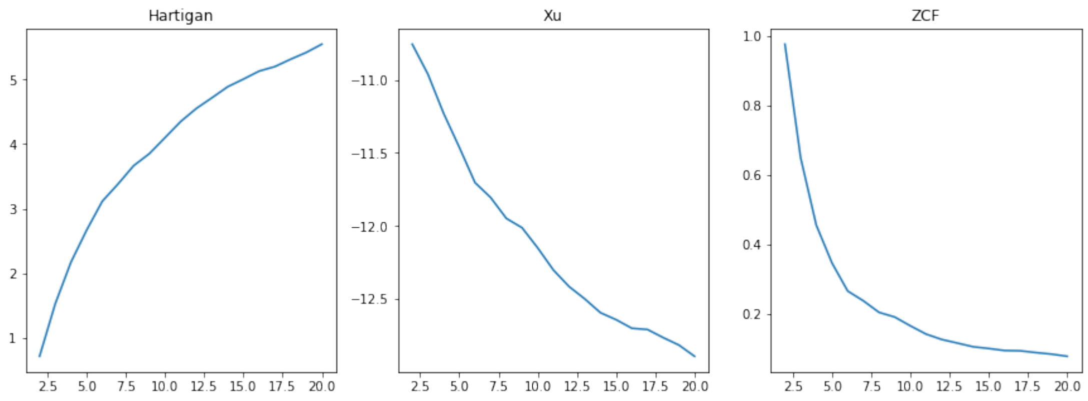
```
AMI= 0.6427148625187993
```

[17]:
```python
lscores = []
nclusters = 20
indices = ['Silhouete', 'CH', 'DB']
for nc in range(2,nclusters+1):
    km = KMeans(n_clusters=nc, n_init=10, random_state=0)
    labels = km.fit_predict(fdata)
    lscores.append((
        silhouette_score(fdata, labels),
        calinski_harabasz_score(fdata, labels),
        davies_bouldin_score(fdata, labels)))

fig = plt.figure(figsize=(15,5))
for i,ind in enumerate(indices):
  ax = fig.add_subplot(1,3,i+1)
  plt.title(ind)
  plt.plot(range(2,nclusters+1), [x[i] for x in lscores]);
```



But results are not much better.

[18]:
```
lscores = []
nclusters = 20
indices = ['Hartigan', 'Xu', 'ZCF']
for nc in range(2,nclusters+1):
    km = KMeans(n_clusters=nc, n_init=10, random_state=0)
    labels = km.fit_predict(fdata)
    lscores.append(scatter_matrices_scores(fdata, labels, indices=␣
 ↪['Hartigan', 'Xu', 'ZCF']))

fig = plt.figure(figsize=(15,5))
for i, ind in enumerate(indices):
  ax = fig.add_subplot(1,3,i+1)
  plt.title(ind)
  plt.plot(range(2,nclusters+1), [x[ind] for x in lscores]);
```



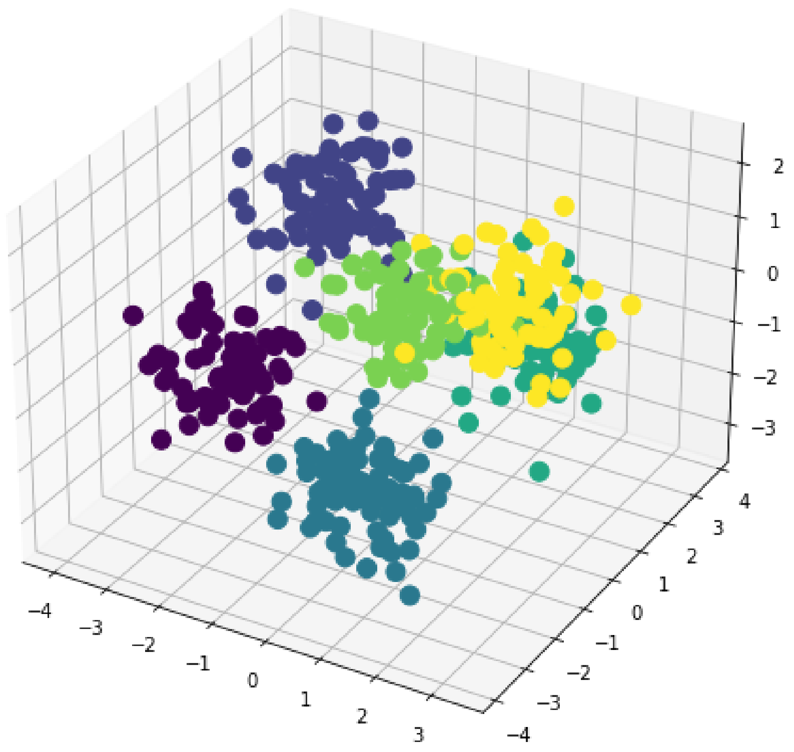
For any of the indices.

3.6 Notebook: External Indices

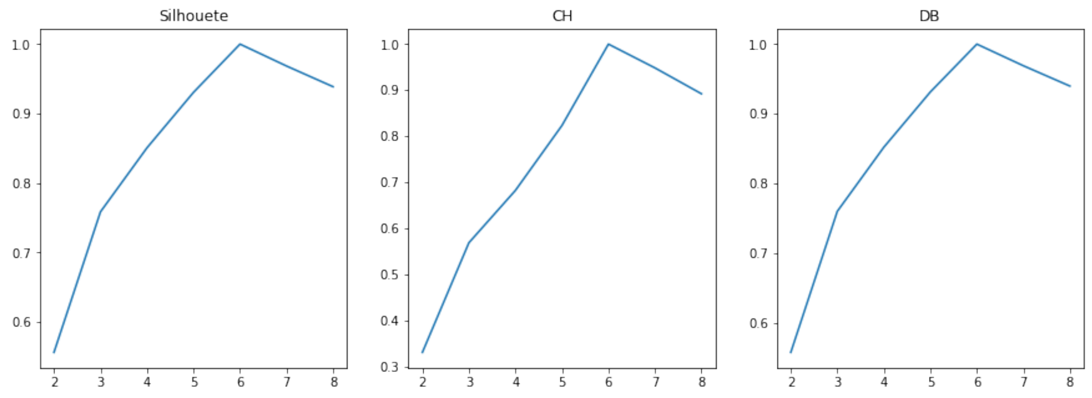
Another approach to cluster validation is to use external indices, this assumes that we know the labels and we compute how “correlated” are to the ones discovered.

[19]:
```
blobs, blabels = make_blobs(n_samples=500, n_features=10, centers=6,␣
 ↪cluster_std=.6, center_box=(-3,3))
fig = plt.figure(figsize=(8,8))
```

```
ax = fig.add_subplot(111, projection='3d')
plt.scatter(blobs[:, 0], blobs[:, 1], zs=blobs[:, 2], depthshade=False,␣
 ↪c=blabels, s=100);
```



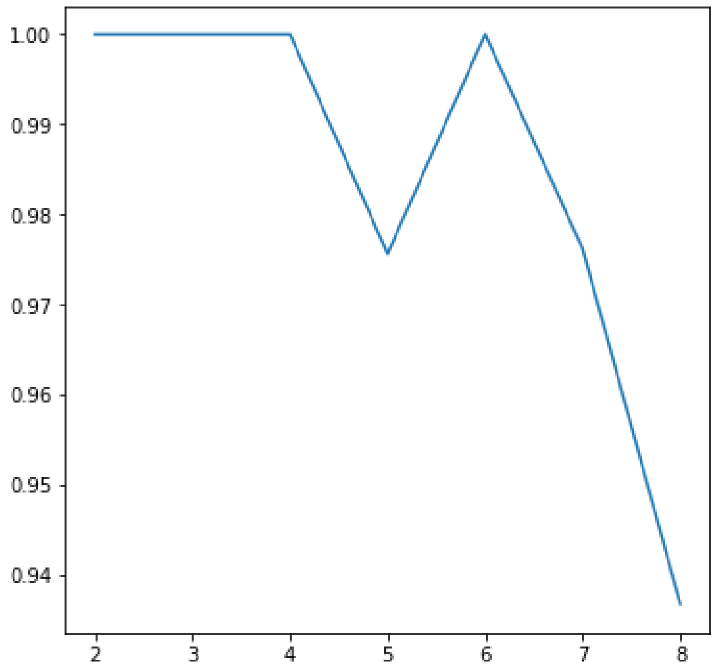
[20]:
```
lscores = []
nclusters = 8
indices = ['Silhouete', 'CH', 'DB']
for nc in range(2,nclusters+1):
    km = KMeans(n_clusters=nc, n_init=10, random_state=0)
    labels = km.fit_predict(blobs)
    lscores.append((
        adjusted_mutual_info_score(blabels, labels),
        adjusted_rand_score(blabels, labels),
        normalized_mutual_info_score(blabels, labels)))

fig = plt.figure(figsize=(15,5))
for i,ind in enumerate(indices):
  ax = fig.add_subplot(1,3,i+1)
  plt.title(ind)
  plt.plot(range(2,nclusters+1), [x[i] for x in lscores]);
```



In this case the maximum value of the scores is at the correct number of clusters

We do not always have a set of labels to compare with, another approach to take advantage of algorithms that return different partitions depending on initialization (e.g. K-means) and test how close are the labelings, the correct number of clusters should be where the different clusters have higher agreement.

[21]:
```
lscores = []
nclusters = 8
for nc in range(2,nclusters+1):
    llabels = []
    for i in range(10):
        km = KMeans(n_clusters=nc, n_init=10)
        labels = km.fit_predict(blobs)
        llabels.append(labels)
    mscores = []
    for i in range(10):
        for j in range(i,10):
            mscores.append(adjusted_mutual_info_score(llabels[i],␣
 ↪llabels[j]))
    lscores.append(np.mean(mscores))

fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111)
plt.plot(range(2,nclusters+1), lscores);
```



In this case there is a range of possible clusters with high agreement.

3.7 Other methods for clustering validation

There are less systematic methods for assessing cluster validity or clusterness, but that can be used as a preliminary step of the study of a dataset. The simplest way to gain some intuition about the characteristics of our data is to try to visualize it and see if there are some natural clusters.

Usually the dimensionality of the data will not allow directly observing the patterns in the data, but the different dimensionality reduction techniques explained in 1.3.1 can be used to project the dataset to 2 or 3 dimensions. Depending on our knowledge about the domain we can decide the more adequate method, or we can begin with the simplest method (e.g.: PCA) and decide how to continue from there depending on the results. The hope is that the clusters that can be observed in the new space could represent clusters in the original space. The confidence on this being the case depends on the reconstruction error of the transformed data and that the transformation maintains the relations in the original space.

Other approach is to visualize the distance matrix of the data. This matrix represents the examples relationships, using different distance functions will generate different relationships, so they have to be chosen carefully. The values of the matrix have to be rearranged, so the closer examples appear in adjacent columns. The apparition of patterns in the rearranged matrix can be a sign of cluster tendency.

There are several methods for rearranging the distance matrix. The simplest one uses a hierarchical clustering algorithm and rearranges the matrix using an in-order traversal of the tree. Results will depend on the algorithm, and the distance/similarity function used. The main advantage of this method is that it can be applied to quantitative and qualitative data, because only the distances are used and not the values of the attributes.

The main problem of these methods is that they are computationally expensive (at least quadratic on time and space). Also, to see patterns in the distance matrix is not always a guarantee of having actual clusters in the data. As an example, figure 3.7 shows the distance matrix of three datasets. The first one corresponds to five well separated clusters, it can be seen

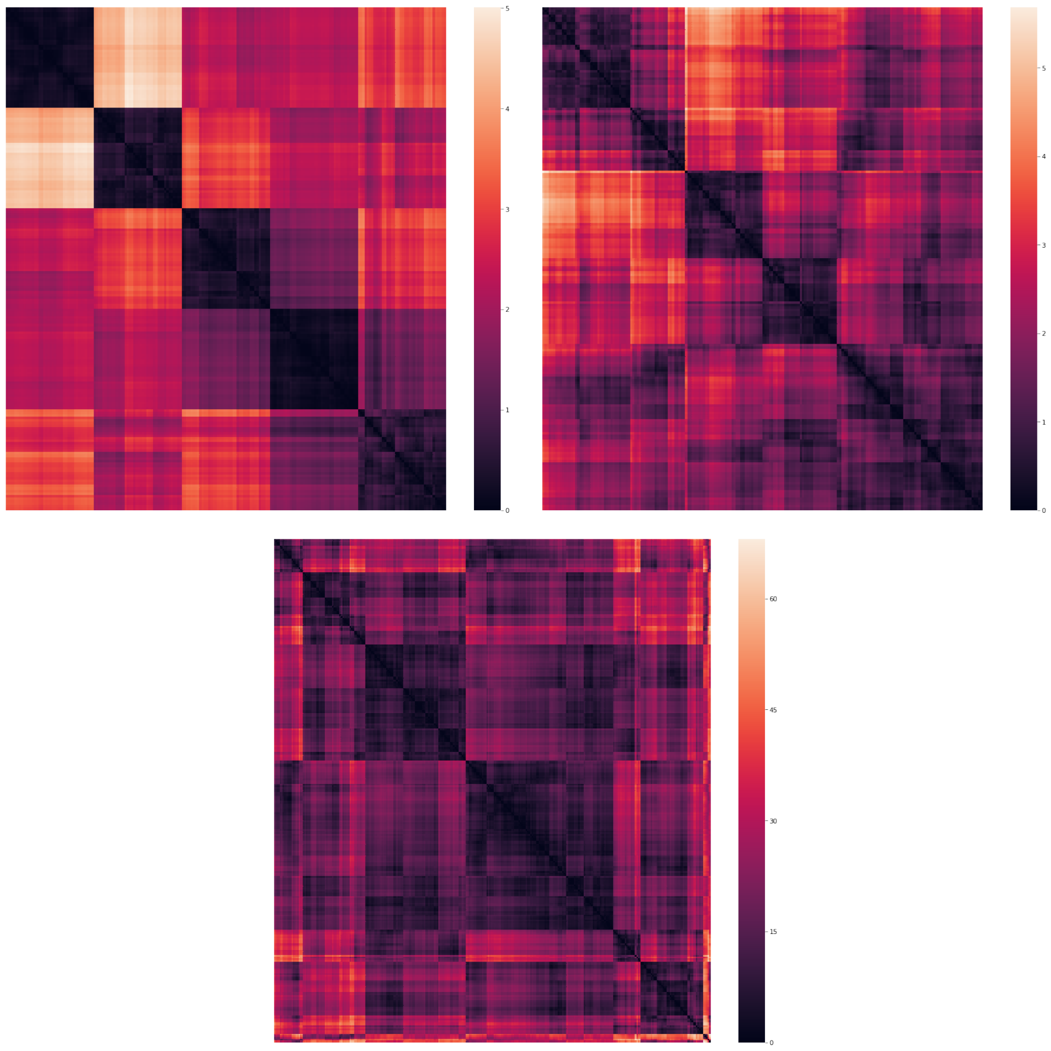


Figure 3.7: Distance matrix visualization. The first plot corresponds to a dataset with 5 well separated clusters, the second one to 5 overlapping clusters and the third one to random generated data

a clear structure in the distance matrix as five squares in the diagonal. The second one is for five overlapped clusters, the overlapping results in fuzzy defined square patterns in the diagonal that makes difficult to appreciate the actual number of clusters. The third one corresponds to a uniform distributed dataset, where no pattern appears in the rearranged distance matrix.

Figure 3.8 shows the distance matrix of three concentric rings dataset using the euclidean and the cosine distances. Being the data non-linearly separable it is difficult to appreciate the actual patterns in the data just by the visualization of the distance matrix, and it can be usually misleading. The matrix using the euclidean distance shows a square pattern corresponding to the inner ring, but the other rings appear like random noise. Using the cosine distance shows a five cluster pattern in the data matrix. This means that one has to be careful when interpreting the results of these methods and has to try different alternatives before making a decision about how to proceed when processing a dataset.

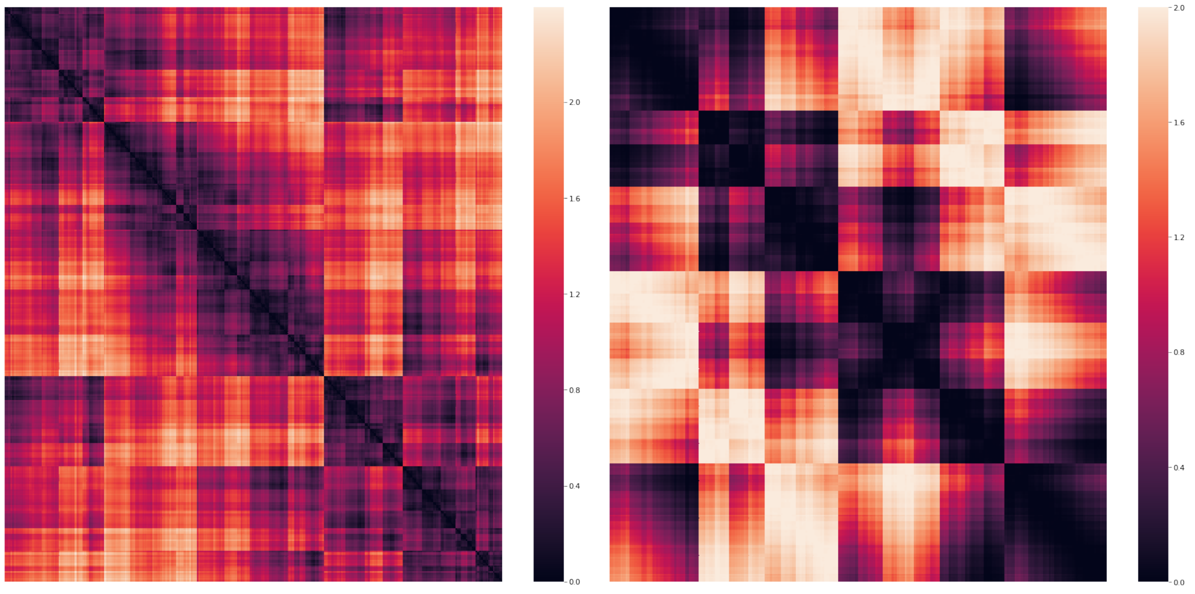


Figure 3.8: Distance matrix visualization for a two concentric circles dataset

3.8 Application: Wheelchair control

This example continues the application described in section 1.6. As a remainder, the domain is about the trajectories recorded from a wheelchair with shared control (patient/computer) for several patients solving different situations. The examples are described by the values obtained from sensors measuring angle/distance to the goal, angle/distance to the nearest obstacle from around the chair (210 degrees).

Now we are asking, given that the visualization of the data (see figure 3.9) has revealed some structure, how many clusters characterize the trajectories? The visualization reveals three clear densities using the tree first components of the PCA, but as commented in the previous section, visualization specially when reducing largely the number of dimensions can be misleading.

To confirm or refine the number of clusters, two different approaches are applied measuring the quality of clusters obtained using the K-means algorithm for different number of clusters. First using the silhouette index as internal validation criteria. Figure 3.10 (first plot) shows the variation of this criterion with the number of clusters. It presents a clear peak at tree clusters, confirming the intuition obtained from the visualization. The second approach used cluster stability as validation method. In this case several clusterings for each number of clusters have been generated and the adjusted rand and NMI indices have been used to compare the clusters. The idea is that if a number of clusters is adequate for the data, the partitions generated will be more similar among them. The second and third plot from figure 3.10 show the evolution of these indices. It can be seen that 2, 3 and 5 have the highest values for both indices. From the visualization of the data it is clear why two clusters has the higher value, the small cluster is very well separated from the rest of the data. Three clusters was expected also from the visualization. Five clusters is also a stability point that is worth exploring using other methods or that can be explained because the larger clusters are evidently not spherical and are consistently divided in half by K-means.

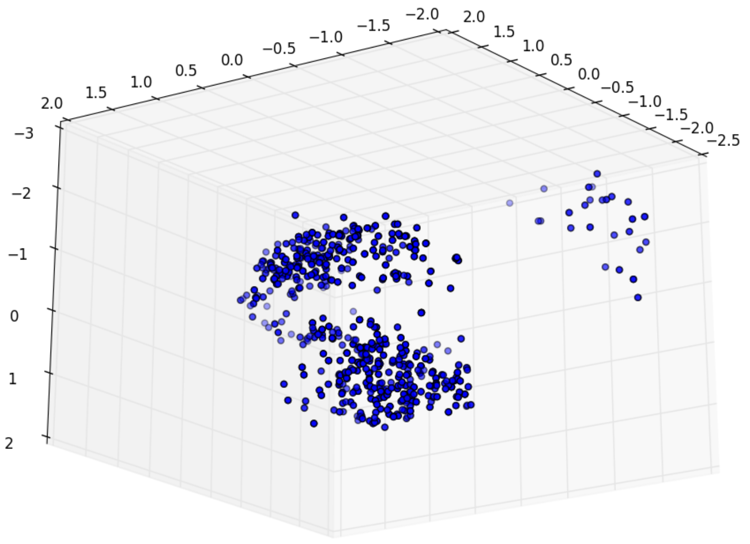


Figure 3.9: Visualization of the three principal components obtained from PCA

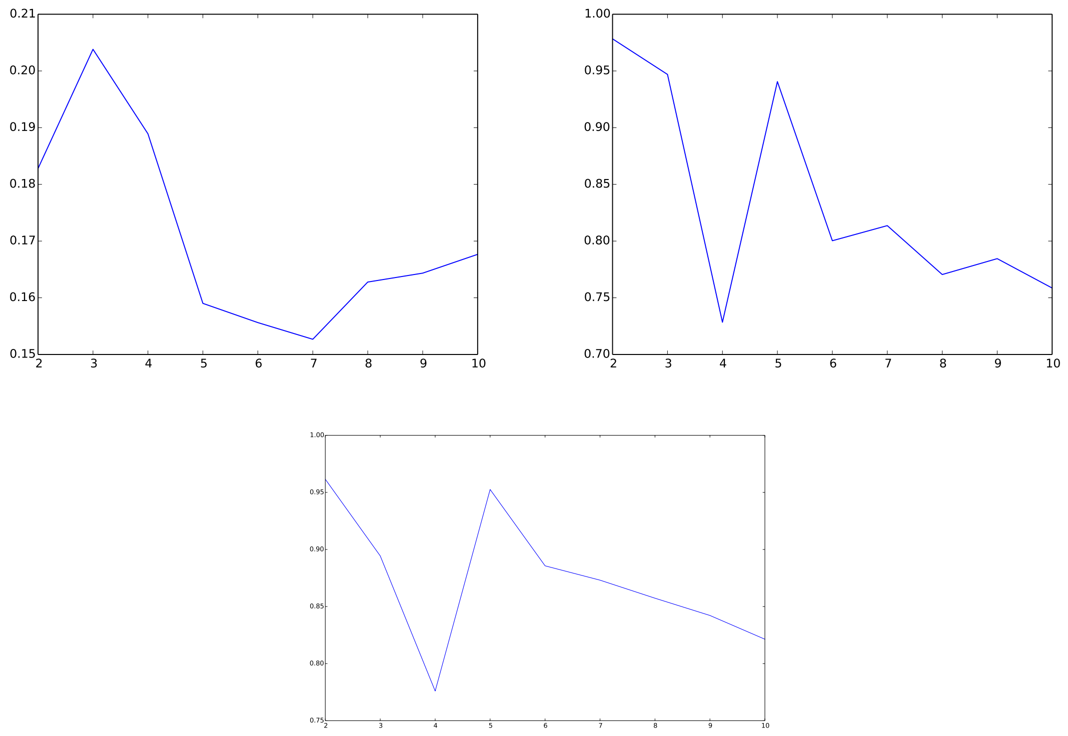


Figure 3.10: Plot of the silhouette, adjusted rand and normalized mutual information indices for different number of clusters for the wheelchair dataset

4. Clustering Large Datasets

4.1 Clustering in Big Data

According to a recent poll about the most frequent tasks and methods employed in data mining projects (KDNuggets, 2019), clustering was the third most frequent task. It is usual that these projects involve areas like astronomy, bioinformatics or finance, that generate large quantities of data. Also, according to a recurrent poll of KDNuggets the most frequent size of the datasets being processed in the range of tens of gigabytes, with a wide distribution ranging from the gigabyte and hundreds of terabytes sizes. It is also common that, in some of these domains, data is a continuous stream representing a boundless dataset, that is collected and processed in batches to incrementally update or refine a previously built model.

The classical methods for clustering (e.g.: K-means, hierarchical clustering) are not able to cope with this increasing amount of data. The reason is mainly because either the constraint of maintaining all the data in main memory or the temporal complexity of the algorithms. This makes them impractical for the purpose of processing these increasingly larger datasets. This means that the need of scalable clustering methods is a real problem and in consequence some new approaches are being developed.

There are several methodologies that have been used to scale clustering algorithms, some inspired in methodologies successfully used for supervised machine learning, other specific for this unsupervised task. For instance, some of these techniques use different kinds of sampling strategies, in order to store in memory only a subset of the data. Others are based on the partition of the whole dataset in several independent batches for separate processing and the merging of the result in a consensuated model. Some methodologies assume that the data is a continuous stream and has to be processed online or in successive batches. Also, these techniques are integrated in different ways depending on the model that is used for the clustering process (prototype based, density based, ...). This large variety of approaches makes necessary to define their characteristics and to organize them in a coherent way.

4.2 Scalability strategies

The strategies used to scale clustering algorithms range from general strategies that can be adapted to any algorithm, to specific strategies that exploit the characteristics of the algorithm in order to reduce its computational cost.

Some strategies are also dependent on the type of data that are used. For instance, only clustering algorithms that incrementally build the partition can be used for data streams. For this kind of datasets it means that the scaling strategy has to assume that the data will be processed continuously and only one pass through the data will be allowed. For applications where the whole dataset can be stored in secondary memory, other possibilities are also available.

The different strategies applied for scalability are not disjoint and several strategies are used in combination. Some of them also allow for a distributed/parallel implementation of the clustering algorithm using for instance map-reduce or peer-to-peer networks. These strategies can be classified in:

One-pass strategies: The constraint assumed is that the data only can be processed once and sequentially. A new example is integrated in the model each iteration. Depending on the type of the algorithm a data structure can be used to efficiently determine how to perform this update. This strategy does not only apply to data streams and can be actually used for any dataset.

Summarization strategies: It is assumed that all the examples in the dataset are not needed for obtaining the clustering, so an initial preprocess of the data can be used to reduce its size by combining examples. The preprocess results in a set of representatives of groups of examples that fits in memory. The representatives are then processed by the clustering algorithm.

Sampling/batch strategies: It is assumed that processing samples of the dataset that fit in memory allows obtaining an approximation of the partition of the whole dataset. The clustering algorithm generates different partitions that are combined iteratively to obtain the final partition.

Approximation strategies: It is assumed that certain computations of the clustering algorithm can be approximated or reduced. These computations are mainly related with the distances among examples or among the examples and the cluster prototypes.

Divide and conquer strategies: It is assumed that the whole dataset can be partitioned in roughly independent datasets and that the combination/union of the results for each dataset approximates the true partition.

4.2.1 One-pass strategies

The idea of this strategy is to reduce the number of scans of the data to only one. This constraint may be usually forced by the circumstance that the dataset can not fit in memory and it has to be obtained from disk. Also, the constraint could be imposed by a continuous process that does not allow storing all the data before processing it.

Sometimes this strategy is used to perform a preprocess of the dataset. This results in two stages algorithms, a first one that applies the one-pass strategy and a second one that process in memory a summary of the data obtained by the first stage.

The assumption of the first stage is that a simple algorithm can be used to obtain a coarse representation of the clusters in the data and that this information will be enough to partition the whole dataset.

Commonly this strategy is implemented using the leader algorithm. This algorithm does not provide very good clusters, but can be used to estimate densities or approximate prototypes, reducing the computational cost of the second stage.

4.2.2 Summarization Strategies

The purpose of this strategy is to obtain a coarse approximation of the data without losing the information that represent the different densities of examples. This summarization strategy assumes that there is a set of sufficient quantities that can be computed from the data, capable of representing their characteristics. For instance, by using sufficient statistics like mean and variance.

The summarization can be performed single level, as a preprocess that is feed to a cluster algorithm able to process summaries instead of raw examples, or also can be performed in a hierarchical fashion. This hierarchical scheme can reduce the computational complexity by using a multi level clustering algorithm or can be used as an element of a fast indexing structure that reduces the cost of obtaining the first level summarization.

4.2.3 Sampling/batch strategies

The purpose of sampling and batch strategies is to allow performing the processing in main memory for a part of the dataset.

Sampling assumes that only a random subset or subsets of the data are necessary to obtain the model for the data and that the complete dataset is available from the beginning. The random subsets can be or not disjoint. If more than one sample of the data is processed, the successive samples are integrated with the current model. This is usually done using an algorithm able to process raw data and cluster summaries. The algorithms that use this strategy do not process all the data, so they scale on the size of the sampling and not on the size of the whole dataset.

The use of batches assumes that the data can be processed sequentially and that after applying a clustering algorithm to a batch, the result can be merged with the results from previous batches. This processing assumes that data is available sequentially as in a data stream and that the batch is complete after observing an amount of data that fits in memory.

4.2.4 Approximation strategies

These strategies assume that some computations can be saved or approximated with reduced or null impact on the final result. The actual approximation strategy is algorithm dependent, but usually the most costly part of clustering algorithms corresponds to distance computation among instances or among instances and prototypes. This circumstance focus these strategies particularly on hierarchical, prototype based and some density based algorithms, because they use distances to decide how to assign examples to partitions.

For example, some of these algorithms are iterative and the decision about what partition is assigned to an example does not change after a few iterations. If this can be determined at an early stage, all these distance computations can be avoided in successive iterations.

This strategy is usually combined with a summarization strategy where groups of examples are reduced to a point that is used to decide if the decision can be performed using only that point or the distances to all the examples have to be computed.

4.2.5 Divide and conquer strategies

This is a general strategy applied in multiple domains. The principle is that data can be divided in multiple independent datasets and that the clustering results can be then merged on a final model. This strategy sometimes rely on a hierarchical scheme to reduce the computational cost of merging all the independent models. Some strategies assume that each independent clustering represent a view of the model, being the resulting clustering a consensus of partitions. The approach can also result on almost independent models that have to be joined, in this case the problem to solve is how to merge the parts of the models that represent the same clusters.

4.3 Algorithms

All these scalability strategies have been implemented in several algorithms that represent the full range of different approaches to clustering. Usually more than one strategy is combined in an algorithm to take advantage of the cost reduction and scalability properties. In this section, a review of a representative set of algorithms and the use of these strategies is presented.

4.3.1 Scalable hierarchical clustering

The main drawback of hierarchical clustering is its high computational cost (time $O(n^2)$, space $O(n^2)$) that makes it impractical for large datasets. The proposal in [109] divides the clustering process in two steps. First a one pass clustering algorithm is applied to the dataset, resulting in a set of cluster summaries that reduce the size of the dataset. This new dataset fits in memory and can be processed using a single link hierarchical clustering algorithm.

For the one-pass clustering step, the leader algorithm is used. This algorithm has as parameter (d), the maximum distance between example and cluster prototype. The processing of each example follows the rule, if the nearest existing prototype is closer than d, it is included in that cluster and its prototype recomputed, otherwise, a new cluster with the example is created. The value of the parameter is assumed to be known or can be estimated from a sample of the dataset. The time complexity of this algorithm is $O(nk)$ being k the number of clusters obtained using the parameter d.

The first phase of the proposed methodology applies the leader algorithm to the dataset using as a parameter half the estimated distance between clusters (h). For the second stage, the centers of the obtained clusters are merged using the single-link algorithm until the distance among clusters is larger than h.

The clustering obtained this way is not identical to the resulting from the application of the single-link algorithm to the entire dataset. To obtain the same partition, an additional process is performed. During the merging process, the clusters that have pairs of examples at a distance less than h are also merged. For doing this, only the examples of the clusters that are at a distance less than $2h$ have to be examined. The overall complexity of all three phases is $O(nk)$, that corresponds to the complexity of the first step. The single-link is applied only to the cluster obtained by the first phase, reducing its time complexity to $O(k^2)$, being thus dominated by the time of the leader algorithm.

4.3.2 CURE

CURE [55] is a hierarchical agglomerative clustering algorithm. The main difference with the classical hierarchical algorithms is that it uses a set of examples to represent the clusters, allowing for non-spherical clusters to be represented. It also uses a parameter that shrinks the representatives towards the mean of the cluster, reducing the effect of outliers and smoothing the shape of the clusters. Its computational cost is $O(n^2 \log(n))$

The strategy used by this algorithm to attain scalability combines a divide an conquer and a sampling strategy. The dataset is first reduced by using only a sample of the data. Chernoff bounds are used to compute the minimum size of the sample so it represents all clusters and

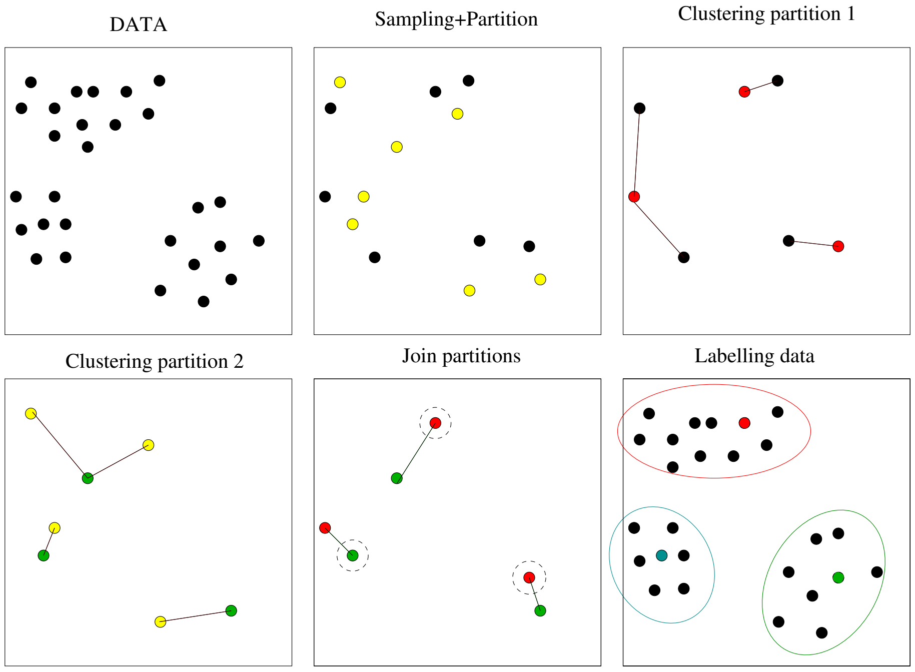


Figure 4.1: CURE

approximates adequately their shapes.

In the case that the minimum size of the sample does not fit in memory a divide and conquer strategy is used. The sample is divided in a set of disjoint batches of the same size and clustered until a certain number of clusters is achieved or the distance among clusters is less than an specified parameter. This step has the effect of a pre-clustering of the data. The clusters representatives from each batch are merged, and the same algorithm is applied until the desired number of clusters is achieved. A representation of this strategy appears in figure 4.1. Once the clusters are obtained all the dataset is labeled according to the nearest cluster. The complexity of the algorithm is $O(\frac{n^2}{p}\log(\frac{n}{p}))$, being n the size of the sample and p the number of batches used.

4.3.3 Scalable K-means

This early algorithm for clustering scalability presented in [14] combines a sampling strategy and a summarization strategy. The main purpose of this algorithm is to provide an online and anytime version of the K-means algorithm that works with a pre-specified amount of memory.

The algorithm repeats the following cycle until convergence:

1. Obtain a sample of the dataset that fits in the available memory
2. Update the current model using K-means
3. Classify the examples as:
    (a) Examples needed to compute the model
    (b) Examples that can be discarded
    (c) Examples that can be compressed using a set of sufficient statistics as fine grain prototypes

The discarding and compressing of part of the new examples allows reducing the amount of data needed to maintain the model each iteration.

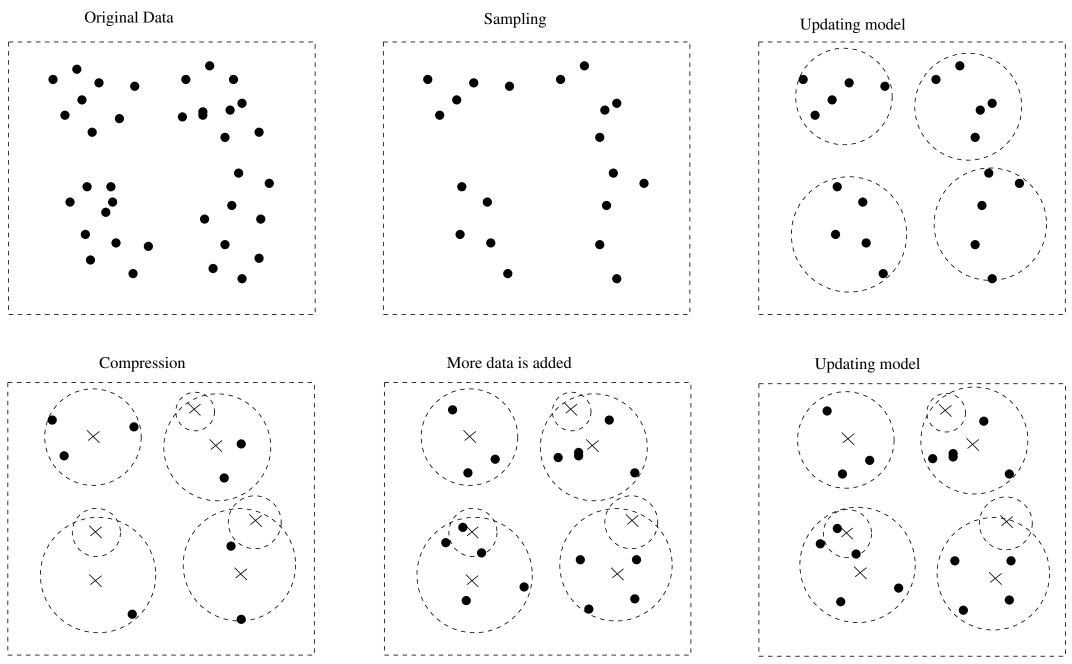


Figure 4.2: Scalable K-means

The algorithm divides the compression of data in two differentiated strategies. The first one is called primary compression, that aims to detect those examples that can be discarded. Two criteria are used for this compression, the first one determines those examples that are closer to the cluster centroid than a threshold. These examples are not probably going to change their assignment in the future. The second one consist in perturbing the centroid around a confidence interval of its values. If an example does not change its current cluster assignment, it is considered that future modifications of the centroid will still include the example.

The second strategy is called secondary compression, that aims to detect those examples that can not be discarded but form a compact subcluster. In this case, all the examples that constitute these compact subclusters are summarized using a set of sufficient statistics. The values used to compute that sufficient statistics are the same used by BIRCH to summarize the dataset.

The algorithm used for updating the model is a variation of the K-means algorithm that is able to treat single instances and also summaries. The temporal cost of the algorithm depends on the number of iterations needed until convergence as in the original K-means, so the computational complexity is $O(kni)$, being k the number of clusters, n the size of the sample in memory, and i the total number of iterations performed by all the updates.

4.3.4 BIRCH

BIRCH [143] is a multi-stage clustering algorithm that bases its scalability in a first stage that incrementally builds a pre-clustering of the dataset. The first stage combines a one pass strategy and a summarization strategy that reduces the actual size of the dataset to a size that fits in memory.

The scalability strategy relies on a data structure named Clustering Feature tree (CF-tree) that stores information that summarizes the characteristics of a cluster. Specifically, the information in a node is the number of examples, the sum of the examples values and the sum of their square values. From these values other quantities about the individual clusters can be computed, for instance, the centroid, the radius of the cluster, its diameter and quantities relative to pairs of clusters, as the inter-cluster distance or the variance increase.

A CF-tree (figure 4.3) is a balanced n-ary tree that contains information that represents

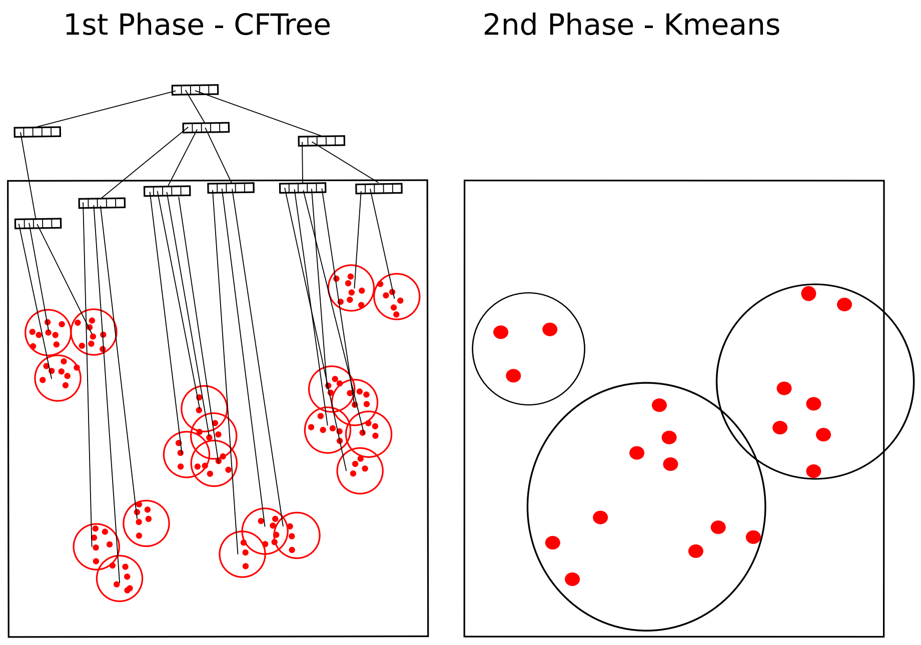


Figure 4.3: BIRCH algorithm

probabilistic prototypes. Leaves of the tree can contain as much as l prototypes and their radius can not be more than t. Each non-terminal node has a fixed branching factor (b), each element is a prototype that summarizes its subtree. The choice of these parameters is crucial, because it determines the actual available space for the first phase. In the case of selecting wrong parameters, the CF-tree can be dynamically compressed by changing the parameters values (basically t). In fact, t determines the granularity of the final groups.

The first phase of BIRCH inserts sequentially the examples in the CF-tree to obtain a set of clusters that summarizes the data. For each instance, the tree is traversed following the branch of the nearest prototype of each level, until a leave is reached. Once there, the nearest prototype from the leave to the example is determined. The example could be introduced in this prototype or a new prototype could be created, depending on whether the distance is greater or not than the value of the parameter t. If the current leave has not space for the new prototype (already contains l prototypes), the algorithm proceeds to create a new terminal node and to distribute the prototypes among the current node and the new leaf. The distribution is performed choosing the two most different prototypes and dividing the rest using their proximity to these two prototypes. This division will create a new node in the ascendant node. If the new node exceeds the capacity of the father, it will be split and the process will continue upwards until the root of the tree is reached if necessary. Additional merge operations after completing this process could be performed to compact the tree.

For the next phase, the resulting prototypes from the leaves of the CF tree represent a coarse vision of the dataset. These prototypes are used as the input of a clustering algorithm. In the original algorithm, single link hierarchical clustering is applied, but also K-means clustering could be used. The last phase involves labeling the whole dataset using the centroids obtained by this clustering algorithm. Additional scans of the data can be performed to refine the clusters and detect outliers.

The actual computational cost of the first phase of the algorithm depends on the chosen parameters. Chosen a threshold t, considering that s is the maximum number of leaves that the CF-tree can contain, also that the height of the tree is $\log_b(s)$ and that at each level b nodes have to be considered, the temporal cost is $O(nb\log_b(s))$. The temporal cost of clustering the leaves of the tree depends on the algorithm used, for hierarchical clustering it is $O(s^2)$. Labeling the

Algorithm 4.1 Mini Batch K-Means algorithm

```
Given: k, mini-batch size b, iterations t, data set X
Initialize each c ∈ C with an x picked randomly from X
v ← 0
for $i$ ← $1$ to $t$ do
    M ← b examples picked randomly from X
    for $x \in M$ do
        d[x] ← f(C,x)
    end
    for $x \in M$ do
        c ← d[x]
        v[c] ← v[c] + 1
        $\eta \leftarrow \frac{1}{v[c]}$
        c ← (1-$\eta$)c+$\eta$x
    end
end
```

dataset has a cost $O(nk)$, being k the number of clusters.

4.3.5 Mini batch K-means

Mini Batch K-means [121] uses a sampling strategy to reduce the space and time that K-means algorithm needs. The idea is to use small bootstrapped samples of the dataset of a fixed size that can be fit in memory. Each iteration, the sample is used to update the clusters. This procedure is repeated until the convergence of the clusters is detected or a specific number of iterations is reached.

Each mini batch of data updates the cluster prototypes using a convex combination of the attribute values of the prototypes and the examples. A learning rate that decreases each iteration is applied for the combination. This learning rate is the inverse of number of examples that have been assigned to a cluster during the process. The effect of new examples is reduced each iteration, so convergence can be detected when no changes in the clusters occur during several consecutive iterations. A detailed implementation is presented in algorithm 4.1.

The mayor drawback of this algorithm is that the quality of the clustering depends on the size of the batches. For very large datasets, the actual size for a batch that can be fit in memory can be very small compared with the total size of the dataset. The mayor advantage is the simplicity of the approach. This same strategy is also used for scaling up other algorithms as for example backpropagation in artificial neural networks.

The complexity of the algorithm depends on the number of iterations needed for convergence (i), the size of the samples (n), and the number of clusters (k) so it is bound by $O(kni)$.

4.3.6 Canopy clustering

Canopy clustering [91] uses a divide and conquer and an approximation strategies to reduce the computational cost. It also uses a two phases clustering algorithm, that is implemented using the well known mapreduce paradigm for concurrent programming.

The first stage divides the whole dataset in a set of overlapping batches called *canopies*. The computation of these batches depends on a cheap approximate distance that determines the neighborhood of a central point given two distance thresholds. The smaller distance (T_2) determines the examples that will belong exclusively to a canopy. The larger distance (T_1)

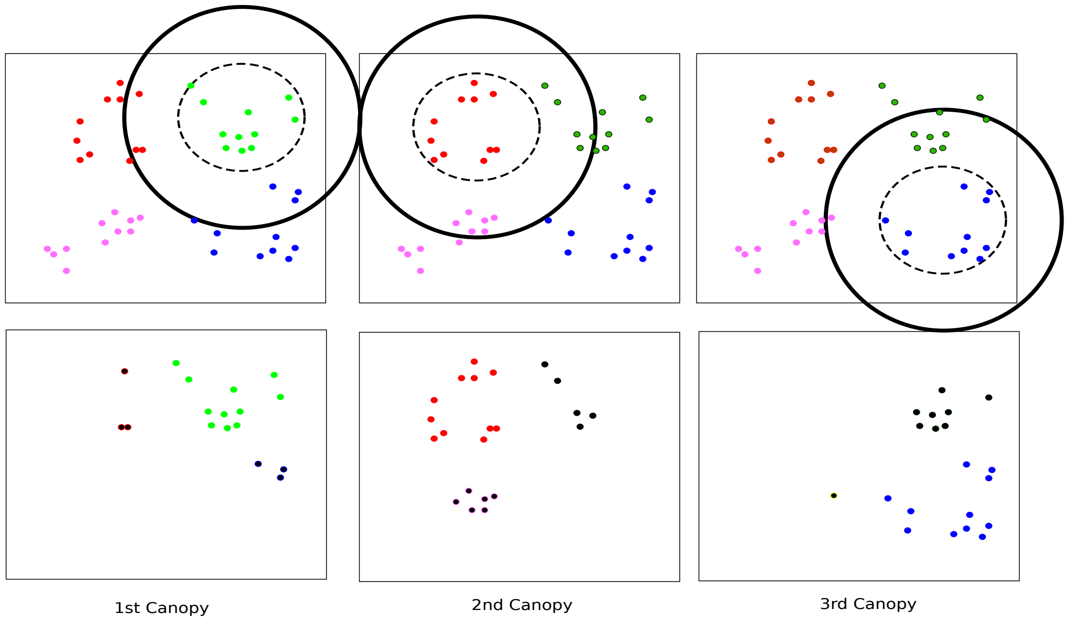


Figure 4.4: Canopy clustering

determines the examples that can be shared with other canopies. The values of these two distance thresholds can be manually determined or computed using crossvalidation.

To actually reduce the computational cost of distance computation the distance function used in this first phase should be cheap to compute. The idea is to obtain an approximation of the densities in dataset. The specific distance depends on the characteristics of the attributes in the dataset, but it is usually simple to obtain such a function by value discretization or using locality sensitive hashing.

The computation of the canopies proceeds as follows: One example is randomly picked as the center of a canopy from the dataset, all the examples that are at a distance less than T_2 are assigned to this canopy and can not be used as centers in the future iterations. All the examples that are at a distance less than T_1 are included in the canopy but can be used as centers in the future. The process is repeated until all the examples have been assigned to a canopy. In figure 4.4 can be seen a representation of this process.

The second stage of the algorithm consist in clustering all the canopies separately. For this process, different algorithms can be used, for example agglomerative clustering, expectation maximization (EM) for gaussian mixtures or K-means. Also different strategies can be used for applying these algorithms. For example, for K-means or EM the number of prototypes for a canopy can be fixed at the beginning, using only the examples inside the canopy to compute them, saving this way many distance computations. Other alternative is to decide the number of prototypes globally, so they can move among canopies and be computed not only using the examples inside a canopy, but also using the means of the nearest canopies.

These different alternatives make difficult to give a unique computational complexity for all the process. For the first stage, the data has to be divided in canopies, this computational cost depends on the parameters used. The method used for obtaining the canopies is similar to the one used by the leader algorithm, this means that equivalently as was shown in 4.3.9, the number of partitions obtained does not depend on the total number of examples (n), but on the volume defined by the attributes and the value of parameter T_2. Being k this number of canopies, the computational cost is bounded by $O(nk)$. The cost of the second stage depends on the specific algorithm used, but the number of distance computations needed for a canopy will be reduced in a factor $\frac{n}{k}$, so for example if single link hierarchical clustering is applied the total

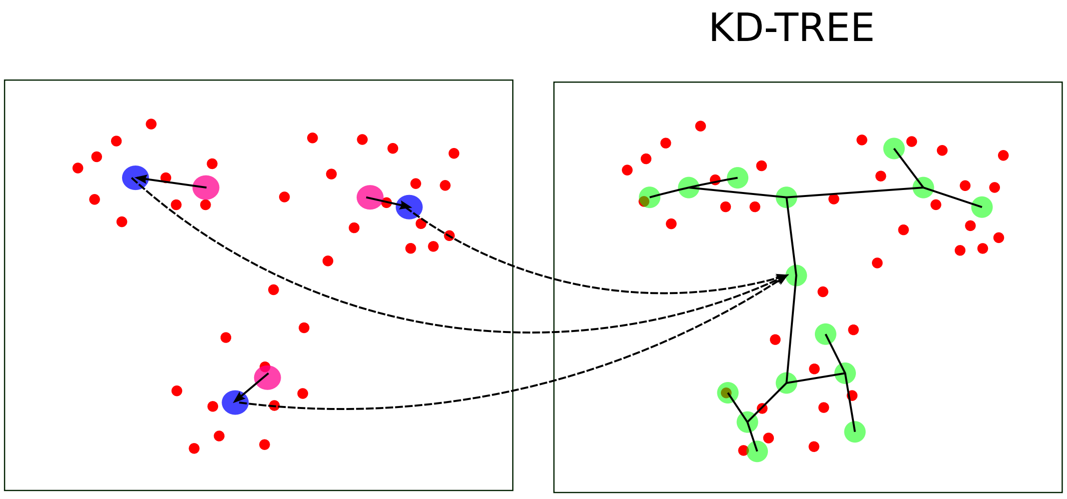


Figure 4.5: Indexed K-means

computational cost of applying this algorithm to k canopies will be $O(\frac{n}{k}^2)$.

4.3.7 Indexed K-means

The proposal in [73] relies on an approximation strategy. This strategy is applied to the K-means algorithm. One of the computations that have most impact in the cost of this algorithm is that, each iteration all the distances from the examples to the prototypes have to be computed. One observation about the usual behavior of the algorithms is that, after some iterations, most of the examples are not going to change their cluster assignment for the remaining iterations, so computing their distances increases the cost, without having an impact in the decisions of the algorithm.

The idea is to reduce the number of distance computations by storing the dataset in an intelligent data structure that allows to determine how to assign them to the cluster prototypes. This data structure is a kd-tree, a binary search tree that splits the data along axis parallel cuts. Each level can be represented by the centroid of all the examples assigned to each one of the two partitions.

In this proposal, the K-means algorithm is modified to work with this structure. First, a kd-tree is built using all the examples. Then, instead of computing the distance from each example to the prototypes and assigning them to the closest one, the prototypes are inserted in this kd-tree. At each level, the prototypes are assigned to the branch that has the closest centroid. When a branch of the tree has only one prototype assigned, all the examples in that branch can be assigned directly to that prototype, avoiding further distance computations. When a leave of the kd-tree is reached and still there is more than one prototype, the distances among the examples and the prototypes are computed, and the assignments are decided by the closest prototype as in the standard K-means algorithm. A representation of this algorithm can be seen in figure 4.5.

The actual performance depends on how separated are the clusters in the data and the granularity of the kd-tree. The more separated the clusters are, the less distance computations have to be performed, as the prototypes will be assigned quickly to only one branch near to the root of the kd-tree.

The time computational cost in the worst case scenario is the same as K-means, as in this case all the prototypes will be assigned to all branches, so all distance computations will be performed. The more favorable case will be when the clusters are well separated and the number of levels in the kd-tree is logarithmic respect to the dataset size (n), this cost will depend also

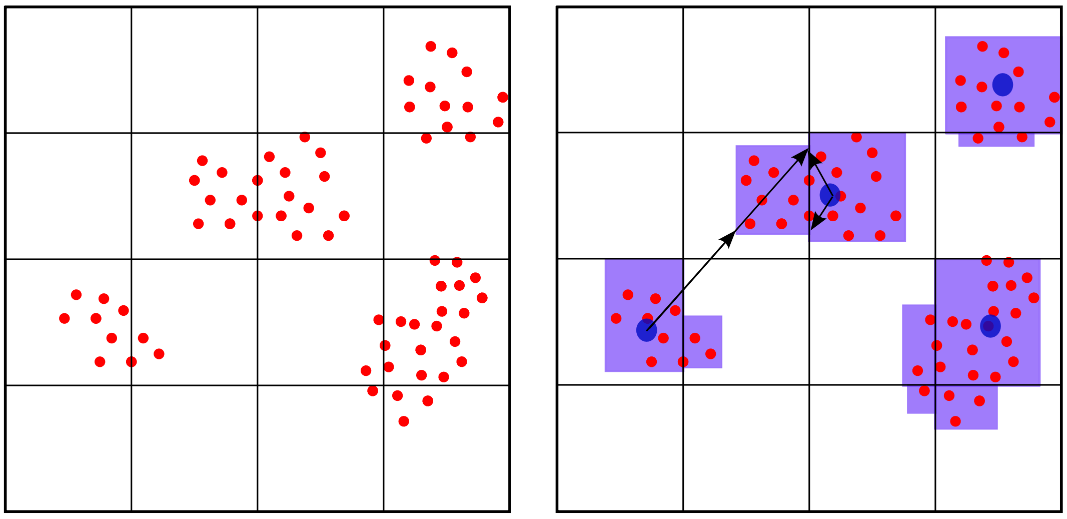


Figure 4.6: Quantized K-means

on the volume enclosed in the leaves of the kd-tree and the number of dimensions (d). The computational cost for each iteration is bound by $\log(2^d k \log(n))$.

The major problem of this algorithm is that as the dimensionality increases, the benefit of the kd-tree structure degrades to a lineal search. This is a direct effect of the curse of the dimensionality and the experiments show that for a number of dimensions larger than 20 there are no time savings.

4.3.8 Quantized K-means

The proposal in [139] relies on an approximation strategy combined with a summarization strategy. The idea is to approximate the space of the examples by assigning the data to a multidimensional histogram. The bins of the histograms can be seen as summaries. This reduces the distance computations by considering all the examples inside a bin of the histogram as a unique point.

The quantization of the space of attributes is obtained by fixing the number of bins for each dimension to $\rho = \lfloor \log_m(n) \rfloor$, being m the number of dimensions and n the number of examples. The size of a bin is $\lambda_l = \frac{\overline{p_l} - \underline{p_l}}{\rho}$, being $\overline{p_l}$ and $\underline{p_l}$ the maximum and minimum value of the dimension l. All examples are assigned to a unique bin depending on the values of their attributes.

From these bins, a set of initial prototypes are computed for initializing a variation of the K-means algorithm. The computation of the initial prototypes uses the assumption that the bins with a higher count of examples are probably in the areas of more density of the space of examples. A max-heap is used to obtain these highly dense bins. Iteratively, the bin with the larger count is extracted from the max-heap and all the bins that are neighbors of this bin are considered. If the count of the bin is larger than its neighbors, it is included in the list of prototypes. All neighbor cells are marked, so they are not used as prototypes. This procedure is repeated until k initial bins are selected. The centroids of these bins are used as initial prototypes.

For the cluster assignment procedure two distance functions are considered involving the distance from a prototype to a bin. The minimum distance from a prototype to a bin is computed as the distance to the nearest corner of the bin. The maximum distance from a prototype to a bin is computed as the distance to the farthest corner of the bin. In figure 4.6, the quantization of the dataset and these distances are represented.

Each iteration of the algorithm first computes the maximum distance from each bin to the prototypes and then it keeps the minimum of these distances as $\overline{d}(b_i, s_*)$. Then, for each prototype the minimum distance to all the bins is computed and the prototypes that are at a distance less than $\overline{d}(b_i, s_*)$ are assigned to the bins.

If only one prototype is assigned to a bin, then all its examples are assigned to the prototype without more distance computations. If there is more than one prototype assigned, the distance among the examples and the prototypes are computed and the examples are assigned to the nearest one. After the assignment of the examples to prototypes, the prototypes are recomputed as the centroid of all the examples.

Further computational improvement can be obtained by calculating the actual bounds of the bins, using the maximum and minimum values of the attributes of the examples inside a bin. This allows to obtain a more precise maximum and minimum distances from prototypes to bins, reducing the number of prototypes that are assigned to a bin.

It is difficult to calculate the actual complexity of the algorithm because it depends on the quantization of the dataset and how separated the clusters are. The initialization step that assigns examples to bins is $O(n)$. The maximum number of bins is bounded by the number of examples n, so at each iteration in the worst case scenario $O(kn)$ computations have to be performed. In the case that the data presents well separated clusters, many bins will be empty, reducing the actual number of computations.

4.3.9 Rough-DBSCAN

In [133] a two steps algorithm is presented. The first step applies a one pass strategy using the leader algorithm, just like the algorithm in the previous section. The application of this algorithm results in an approximation of the different densities of the dataset. These densities are used in the second step, that consists in a variation of the density based algorithm DBSCAN.

This method uses a theoretical result that bounds the maximum number of leaders obtained by the leader algorithm. Given a radius τ and a closed and bounded region of space determined by the values of the features of the dataset, the maximum number of leaders k is bounded by:

$$k \leq \frac{V_S}{V_{\tau/2}} \tag{4.1}$$

being V_S the volume of the region S and $V_{\tau/2}$ the volume of a sphere of radius $\tau/2$. This number is independent of the number of examples in the dataset and the data distribution.

For the first step, given a radius τ, the result of the leader algorithm is a list of leaders ($\mathcal{L}$), their followers and the count of their followers. The second step applies the DBSCAN algorithm to the set of leaders given an ϵ and a $MinPts$ parameters.

The count of followers is used to estimate the count of examples around a leader. Different estimations can be derived from this count. First it is defined $\mathcal{L}_l$ as the set of leaders at a distance less or equal than ϵ to the leader l:

$$\mathcal{L}_l = \{l_j \in \mathcal{L} \mid \|l_j - l\| \leq \epsilon\} \tag{4.2}$$

The measure $roughcard(N_\epsilon(l, \mathcal{D}))$ is defined as:

$$roughcard(N_\epsilon(l, \mathcal{D})) = \sum_{l_i \in \mathcal{L}_l} count(l_i) \tag{4.3}$$

approximating the number of examples less than a distance ϵ to a leader. Alternate counts can be derived as upper and lower bounds of this count using $\epsilon + \tau$ (upper) or $\epsilon - \tau$ (lower) as distance.

Algorithm 4.2 OptiGrid algorithm

```
Given: number of projections k, number of cutting planes q, min cutting quality min_c_q,
       data set X
Compute a set of projections $P = \{P_1, ..., P_k\}$
Project the dataset X wrt the projections $\{P_1(X), ..., P_k(X)\}$
BestCuts ← ∅, Cut ← ∅
for $i \in 1..k$ do
    Cut ← ComputeCuts($P_i(X)$)
    for c in Cut do
        if CutScore(c)> min_c_q then BestCuts.append(c)
    end
end
if BestCuts.isEmpty() then return X as a cluster
BestCuts ← KeepQBestCuts(BestCuts,q)
Build the grid for the q cutting planes
Assign the examples in X to the cells of the grid
Determine the dense cells of the grid and add them to the set of clusters $C$
foreach cluster $cl \in C$ do
    apply OptiGrid to cl
end
```

From this counts it can be determined if a leader is dense or not. Dense leaders are substituted by their followers, non-dense leaders are discarded as outliers. The final result of the algorithm is the partition of the dataset according to the partition of the leaders.

The computational complexity of this algorithm is for the first step $O(nk)$, being k the number of leaders, that does not depend on the number of examples n, but on the radius τ and the volume of the region that contains the examples. For the second step, the complexity of the DBSCAN algorithm is $O(k^2)$, given that the number of leaders will be small for large datasets, the cost is dominated by the cost of the first step.

4.3.10 OptiGrid

OptiGrid [65] presents an algorithm that divides the space of examples in an adaptive multi-dimensional grid that determines dense regions. The scalability strategy is based on recursive divide and conquer. The computation of one level of the grid determines how to divide the space on independent datasets. These partitions can be divided further until no more partitions are possible.

The main element of the algorithm is the computation of a set of low dimensional projections of the data that are used to determine the dense areas of examples. These projections can be computed using PCA or other dimensionality reduction algorithms and can be fixed for all the iterations. For a projection, a fixed number of orthogonal cutting planes are determined from the maxima and minima of the density function computed using kernel density estimation or other density estimation method. These cutting planes are used to compute a grid. The dense cells of the grid are considered clusters at the current level and are recursively partitioned until no new cutting planes can be determined given a quality threshold. A detailed implementation is presented in algorithm 4.2

For the computational complexity of this method. If the projections are fixed for all the computations, the first step can be obtained separately of the algorithm and is added to the total cost. The actual cost of computing the projections depends on the method used. Assuming axis parallel projections the cost for obtaining k projections for N examples is $O(Nk)$, $O(Ndk)$ otherwise, being d the number of dimensions. Computing the cutting planes for k projections can be obtained also in $O(Nk)$. Assigning the examples to the grid depends on the size of the grid and the insertion time for the data structure used to store the grid. For q cutting planes and assuming a logarithmic insertion time structure, the cost of assigning the examples has a cost of $O(Nq\min(q,\log(N)))$ considering axis parallel projections and $O(Nqd\min(q,\log(N)))$ otherwise. The number of recursions of the algorithm is bound by the number of clusters in the dataset that is a constant. Considering that q is also a constant, this gives a total complexity that is bounded by $O(Nd\log(N))$

4.3.11 STREAM LSEARCH

The STREAM LSEARCH algorithm [54] assumes that data arrives as a stream, so holds the property of only examining the data once. The algorithm process the data in batches obtaining a clustering for each batch and merging the clusters when there is no space to store them. This merging is performed in a hierarchical fashion. The strategy of the algorithm is then a combination of one-pass strategy plus batch and summarization strategies.

The basis of the whole clustering scheme is a clustering algorithm that solves the facility location (FL) problem. This algorithm reduces a sequential batch of the data to at most $2k$ clusters, that summarize the data. These clusters are used as the input for the hierarchical merging process. The computational cost of the whole algorithm relies on the cost of this clustering algorithm. This algorithm finds a set of between k and $2k$ clusters that optimizes the FL problem using a binary search strategy. An initial randomized procedure computes the clusters used as initial solution. The cost of this algorithm is $O(nm + nk\log(k))$ being m the number of clusters of the initial solution, n the number of examples and k the number of clusters.

The full algorithm can be outlined as:

1. Input the first m points; use the base clustering algorithm to reduce these to at most $2k$ cluster centroids. The number of examples at each cluster will act as the weight of the cluster.

2. Repeat the above procedure until $\frac{m^2}{2k}$ examples have been processed, so we have m centroids

3. Reduce them to $2k$ second level centroids

4. Apply the same criteria for each existing level so after having m centroids at level i then $2k$ centroid at level $i+1$ are computed

5. After seen all the sequence (or at any time) reduce the $2k$ centroids at top level to k centroids

The number of centroids to cluster is reduced geometrically with the number of levels, so the main cost of the algorithm relies on the first level. This makes the time complexity of the algorithm $O(nk\log(nk))$, while needing only $O(m)$ space.

4.3.12 MapReduce clustering

Some clustering algorithms can be adapted using the MapReduce distributed paradigm. This is easy to do for algorithms where one or more steps make decisions for each example in the dataset independently like in some model based clustering algorithms (K-means, GMM).

In [145] a MapReduce implementation for K-means is described. The mapper function is in charge of deciding the assignment of examples to cluster centers and the reducer function

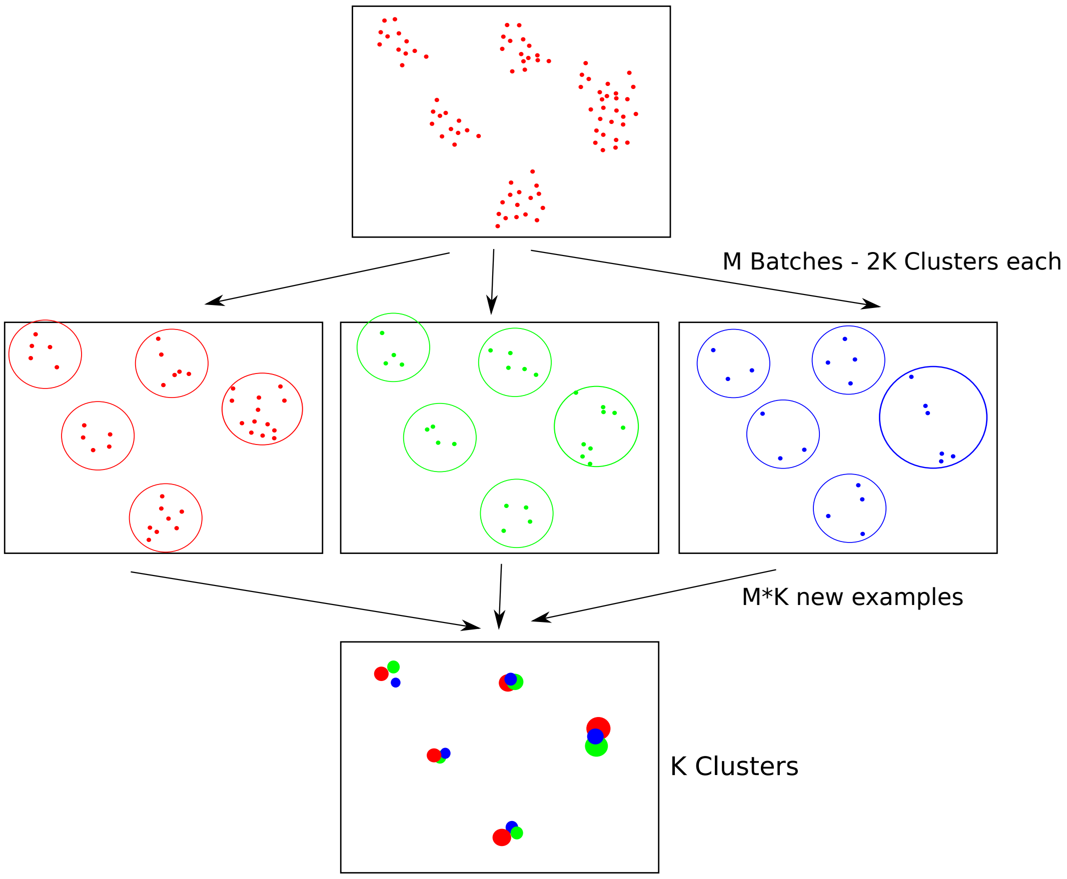


Figure 4.7: STREAM LSEARCH

integrates the examples assigned to a cluster into a new prototype. Each mapper needs a local copy of the current prototypes for computing the assignments. An advantage of this implementation is that the space needed for the assignments and prototype computation is distributed on several machines, allowing processing larger datasets than the non mapreduce version of K-means. Figure 4.8 represents the mapreduce process of a K-means iteration.

This strategy can also be applied to the EM implementation of Gaussian Mixture Models, where the expectation step can be distributed in the mapper functions and the maximization step is implemented in the reducer function. The main difference with respect to the k-means implementation is that the mappers will return a partial assignment for all the examples, so each reducer process all the data.

4.3.13 Peer2Peer clustering

Distributed algorithms over Peer2Peer networks can be used as divide and conquer algorithms that process different chunks of the dataset (as much as fits in memory) and use the network to communicate and combine partial results until all the processes converge to a similar partition. The advantage of using a P2P communication network is to reduce the number of messages passed among the processes and also the possibility of adding and removing processes (data) in the network to adapt for example to changes in the data, to allow continuous processing of new data or even to process multiple simultaneous data streams that can appear or disappear in time.

In [28] the Locally Synchronized P2P K-Means (LSP2P) algorithm is described. The data is distributed in a network of nodes that have only communication with their immediate neighbors (for example in a grid topology) each node receives a set of initial prototypes and a threshold to

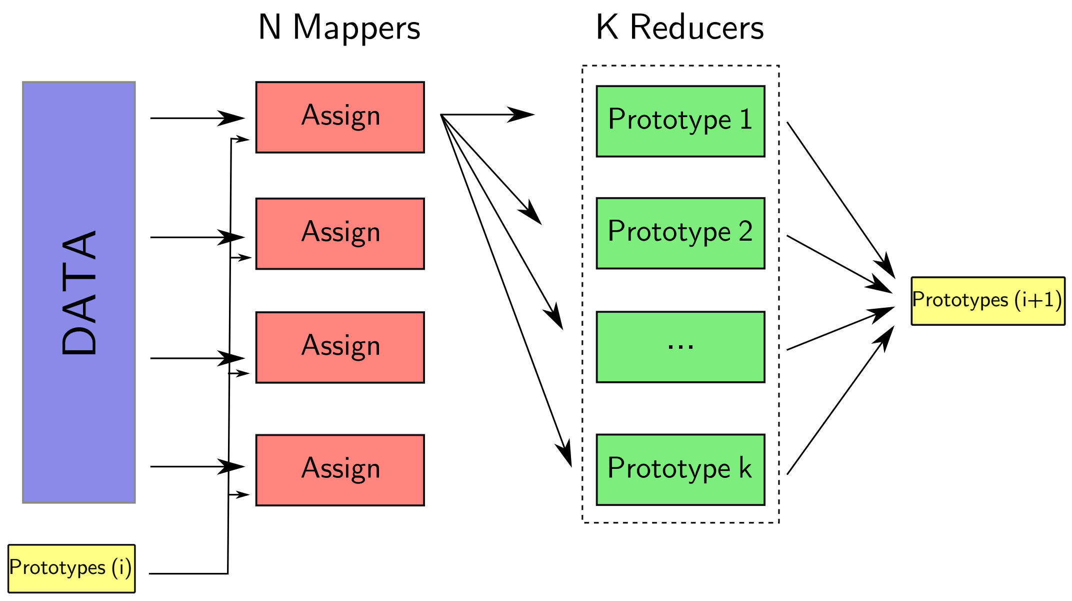


Figure 4.8: MapReduce implementation of a K-means step

be used to determine when the clustering has converged. Given that all nodes receive the same set of prototypes there is no problem determining their correspondence among peers.

All nodes are synchronized and perform an iteration of k-means at a time. Each iteration a node polls its neighbors to obtain their current prototypes and the number of examples at each prototype. The node computes a new set of prototypes as a weighted average of all the prototypes (their own and the ones from its neighbors) and proceeds to the next iteration. If the prototypes received are not much different from the current ones (less than the threshold) the process is terminated. When all the nodes have terminated, they communicate their result to a central node.

The algorithm includes the possibility of adding new nodes. In this case, the new node inquiries its neighbor their number of iterations they have performed and the node iterates until it is synchronized.

In [90] the GDCluster algorithm is described as a decentralized asynchronous distributed algorithm for clustering. This can be adapted for prototype and density based clustering algorithms. The synchronization of the network is based on the *gossiping protocol*. This assumes that periodically pairs of nodes communicate randomly, they interchange messages and integrate the information received in their state. This protocol do not need reliable communication or a stable number of nodes.

Each node p has a dataset D_p and maintains a set of representatives R_p that are computed from its own data and the data received from its peers. The nodes are continuously performing two tasks in parallel. The first one, called *derive* involves a message between two nodes. Each one interchange their data (or a summary of their data if the size is large) and receive a list of examples that are inside a radius ρ of their respective data. The received data is integrated with other data received from other peers. To maintain a limited memory storage the data is integrated as a new set of representatives periodically after a number of peer communications. The computation of the representatives involves the computation of weights to account for the number of examples summarized so it can be used for the clustering process. The collected examples can be also discarded periodically depending on their age to maintain memory constraints. The second task called *collect* involves the interchange of representatives among two nodes. Representatives are merged recomputing their weights and a optional postprocess can eliminate repeated representatives.

To obtain the partition in each node, a clustering algorithm is applied to the representatives using their weights. The continuous interchange of messages among nodes increments the accuracy of the clustering with time, converging to a stable partition with enough iterations.

4.4 Clustering data streams

Data streams are a special case of sequence that models an on-line continuous series of data. The main problem is that the processing of the data has to deal with a dataset that needs theoretically an infinite amount of memory to be stored. It is not possible to see all the data to obtain the model of the data and the applications need to be able to generate a model on-line, that is, it has to be always ready to give an answer to a query.

Each item of the series is an instance that can contain one value, a vector of values, or even be a structured data (think for example of a sequence of texts, like news or tweets). The assumption is that the data is generated from a set of unknown clusters that can be stable in time (this means that the model can be definitive after enough time has passed) or changing over time (there is a drift in the clusters that generate the data). We can have, for instance, a process defined by a set of fixed states that generate the data, or a stream of text that have semantic topics that change over time.

The algorithms for this kind of data have to process the examples incrementally, this means that the model changes with time. This leads to two possible ways to obtain a model from the data. We can keep only the current model, so we are not interested on what happened, or we can store periodic snapshots of the model, so we can provide answers about how the data changes and what has changed.

There are different discovery goals that can be pursued in this context. We can only model the domain to provide answers in the current instant or from the past. We can detect anomalies/novelty/bursts, so the normal behavior is not of interest, but the sudden changes, or the behavior that departs from the normality. Also, the change in the behavior (concept drift) can be modeled, assuming that the process have several stable states that last for long periods. All these goals can be modeled using clustering.

Algorithms for clustering data streams can be defined from four main characteristics:

1. The data structure used to summarize the data
2. The window model used to decide the influence of the current and past data
3. The mechanism for identifying outliers
4. The clustering algorithm used to obtain the partition of the data

Usually the algorithms have two phases, one on-line phase that summarizes the data and decides what and how to keep the information of the stream and an off-line step that is invoked each time a partition of the data is needed.

The data structure used to summarize the data involves the on-line phase. The information that is usually stored includes a set of sufficient statistics able to describe the central tendency and spread of parts of the data. These statistics include the number of observed examples, the linear sum of their values and sum of the square product of their values. From this information it can be derived the cluster mean, its radius and its diameter. These values can be updated incrementally that is and advantage in this scenario and are also additive, so they can be combined for merging sets of examples and clusters.

The specific data structure used to store these statistics is usually hierarchical. This allows working with different granularities of the data. Depending on the space constraints, raw examples can be stored on the leaves or data can be summarized at the bottom levels and discarded. Some structures index the data so the addition of new data can be done efficiently other store prototypes obtained at different time and granularity.

The window model used determines the influence of the current and past data. The sliding window model a fixed time window is decided depending on space and time constraints and only data inside that window is used to update the data structure. The damped window model associates a weight to examples and clusters that gives higher weight to recent data and allows reducing the impact of the past to the results and also allows discarding old data or clusters that do not have updates during a certain amount of time. The landmark window model defines points of interest in time (for example starting of a week) or amount of data since the last landmark to decide what data summarized, discarding the data arrived until the current landmark.

Outlier detection is a difficult task in stream clustering because it is difficult to determine if some example is just an outlier or if more time has to pass to observe examples from a currently low density area of examples. Most of the solutions are derived around the idea of *micro-clusters*. A micro-cluster is a possible dense area of examples that is maintained by the summary structure. The damping window model can be used to determine the prevalence of the data during time, reducing the weight of the micro-cluster with the time. A minimum threshold of density can be defined above which a micro-cluster is determined as representative of a real cluster and also another threshold can be defined that allows to keep not dense enough clusters until it is determined if they are actually outliers.

Respect to the off-line phase that actually yields the partition of the data. In this phase the information collected in the summary structure is passed to the clustering algorithm. The different possibilities are chosen depending on the presumed shape of the data. For spherical clusters, a prototype based clustering algorithm is usually selected like for instance K-means. The summary statistics represents small clusters of examples that are joined. The algorithm is adapted to work with the summary statistics stored. For arbitrary shaped clusters Density based and grid based clustering algorithms are used. The information in the summary structure is treated as a density estimation of the space of examples.

CluStream

CluStream [2] is an algorithm based on an on-line/off-line clustering process able to model a data stream using the concepts of micro-clusters. The idea is to maintain a set of small clusters that evolve in time and describe the process at a specific granularity level. This micro clusters at a specific moment of time can be used to obtain the current clustering of the stream.

The on-line phase maintains/creates micro-clusters using an incremental clustering algorithm similar to the leader algorithm. When new data arrives, it is incorporated to an existing micro-cluster if it is similar enough, or generates a new micro-cluster. Their number is larger than the actual number of clusters to be obtained as model of the data, so their role is to approximate the densities in the data to a certain level of granularity. The number of micro-clusters is fixed, so they are merged to maintain this number and in this way to evolve with the changes in the data. Periodically they are stored to be able to obtain historical snapshots of the data stream.

The off-line phase uses the micro-clusters stored for a window of time. These are merged to obtain a set of micro-clusters that represent the densities during the chosen period. A k-means algorithm is used then to compute the clusters that model the behavior of the stream.

DenStream

DenStream [18] describes a similar approach. It also uses the concept of micro clusters, but in this case they represent a density of examples and the actual examples are discarded with time, avoiding having to store all the data. Each micro cluster contains a set of weighted examples, this

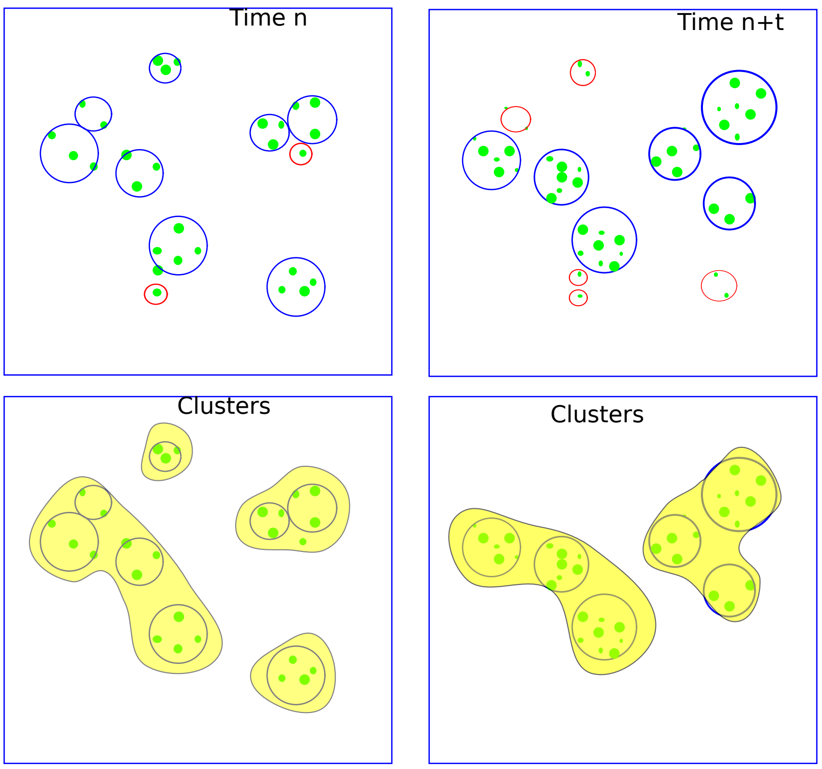


Figure 4.9: Clusterings obtained at different snapshots from the densities defined by micro-clusters

weight fades exponentially with time, that is known as time damping model. These weighted examples are classified according to their surrounding density as core-micro-clusters, that are examples with a weight over a specific threshold, potential-micro-clusters are below that threshold, but above a second threshold, and outlier-micro-clusters, that are examples with a weight below the second threshold. This outlier-micro-clusters disappear from the model after a period.

At a specific moment the model has a set of examples classified on the three sets of micro-clusters. These are used to obtain off-line a clustering of the data using the DBSCAN algorithm. The weights of the examples are used to compute the densities that this algorithm uses to determine the clustering. The main advantage of this model is that the number of cluster to obtain is not predefined, and that the densities defined by the micro-clusters evolve with time because of the fading weights. However, this model only can answer queries about the current instant, because it does not store the evolution of the stream.

Figure 4.9 shows two instants of time of this algorithm, the upper part corresponds to the different micro-clusters with the sizes of the dots representing the weights of the examples. The lower part corresponds to the hypothetical clusters obtained by DBSCAN from these examples.

D-Stream

D-Stream [25] is based on a grid clustering algorithm. The grid is created dynamically mapping examples into the grid and maintaining the density of the different cells. Cells that are not dense enough in a specific point of time can be deleted from the grid. The algorithm assumes that the number of cells is small compared with the cartesian product of the partitions of the attributes. The algorithm uses an exponential decay for reducing the weight of the examples and allowing cells to fade, adapting this way to the changes of the stream.

A set of thresholds distinguish between cells that are dense (normal), sparse (sporadic) and transitional. For each cell, information that allows to recompute the weights of the cell is stored like the time of last updated, the time the cell was marked as sporadic, the density of the cell the label of the cell and information about if the cell is normal or sporadic. The status of the cells are checked at regular intervals (parameter) and checked for change of status.

Given specific thresholds depending on time and properties about the weights related to these times, a cell can be changed to sporadic, and if it is not updated in an interval of time then it is deleted, if it is updated, it is maintained as sporadic if it holds the conditions or it is promoted to normal otherwise.

The clustering process is performed offline and joints cells that have a common border, distinguishing cells that are inside and outside the cluster, the cells that belong to the cluster are the dense ones or the transitional ones.

4.5 Notebook: Scalable Clustering

[1]:
```
from sklearn.cluster import Birch, MiniBatchKMeans, KMeans
import matplotlib.pyplot as plt
import numpy as np
from numpy import loadtxt
import time
from sklearn.metrics import adjusted_mutual_info_score
import warnings
from kemlglearn.datasets import make_blobs

warnings.filterwarnings("ignore")
%matplotlib inline

citypath = '../../Data/' # Adjust the path to your local dataset to run
data = 'LONpos.csv.gz'
citypos = loadtxt(citypath+data, delimiter=',')
nclusters = 201
```

4.5.1 Increasing the number of clusters

Most of the clustering algorithms that we have seen do not scale to large number of examples or even clusters

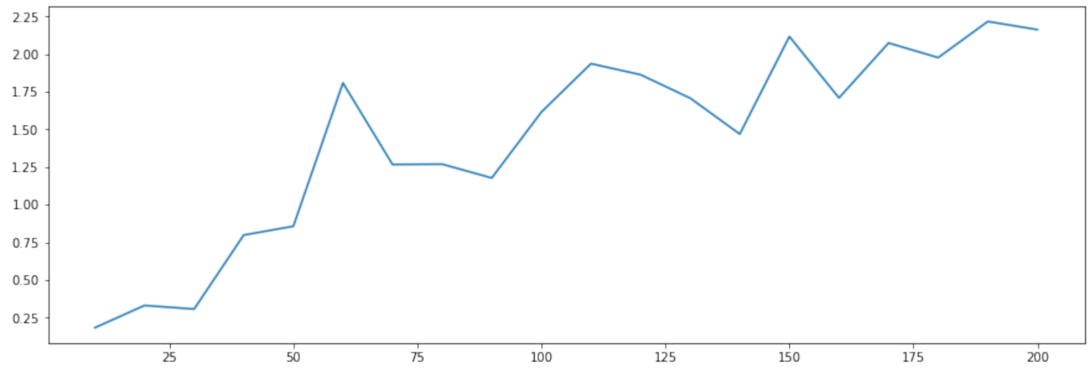
We will play with geographical data of crimes commited in the city of london (over 85.000 examples)

Let's see what happens when we increase the number of clusters

K-means

The cost of K-means corresponds to the update of the centroids. As the number of clusters increases it should scale linearly

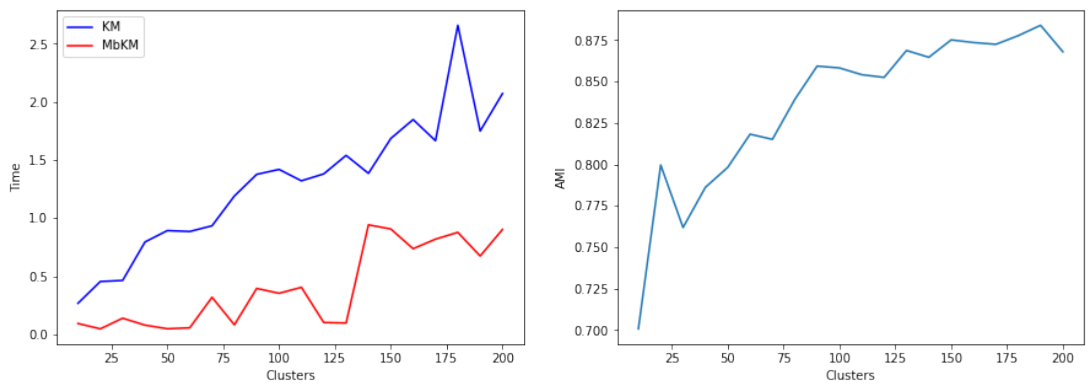
[2]:
```
ltimes = []
for nc in range(10, nclusters, 10):
    km = KMeans(n_clusters=nc, n_init=1)
    itime = time.perf_counter()
    kmlabels = km.fit_predict(citypos)
    etime = time.perf_counter()
    ltimes.append(etime-itime)

fig = plt.figure(figsize=(15,5))
ax = fig.add_subplot(111)
plt.plot(range(10, nclusters, 10), ltimes);
```



Minibatch K-means

Minibatch K-means is an incremental version of K-means that uses random batches of examples to reduce the amount of memory and time. It is designed for datasets that can not be stored in memory, if the clusters are difficult to find the size of the batch could improve the results, if not, small sizes can obtain results good enough in practice.

```
[3]: ltimes = []
ltimeskm = []
agree = []
mblb = []
kmlb = []

for nc in range(10, nclusters, 10):
    mbkm = MiniBatchKMeans(n_clusters=nc, batch_size=500, n_init=1,␣
 ↪max_iter=5000)
    itime = time.perf_counter()
    mblb.append(mbkm.fit_predict(citypos))
    etime = time.perf_counter()
    ltimes.append(etime-itime)

for nc in range(10, nclusters, 10):
    km = KMeans(n_clusters=nc, n_init=1)
    itime = time.perf_counter()
    kmlb.append(km.fit_predict(citypos))
    etime = time.perf_counter()
    ltimeskm.append(etime-itime)

for mbkmlabels, kmlabels in zip(mblb,kmlb):
    agree.append(adjusted_mutual_info_score(mbkmlabels, kmlabels))

fig = plt.figure(figsize=(15,5))
ax = fig.add_subplot(121)
plt.plot(range(10, nclusters, 10), ltimeskm, color='b', label='KM')
plt.plot(range(10, nclusters, 10), ltimes, color='r', label='MbKM')
plt.xlabel('Clusters')
plt.ylabel('Time')
plt.legend()
ax = fig.add_subplot(122)
plt.plot(range(10, nclusters, 10), agree)
plt.xlabel('Clusters')
plt.ylabel('AMI');
```

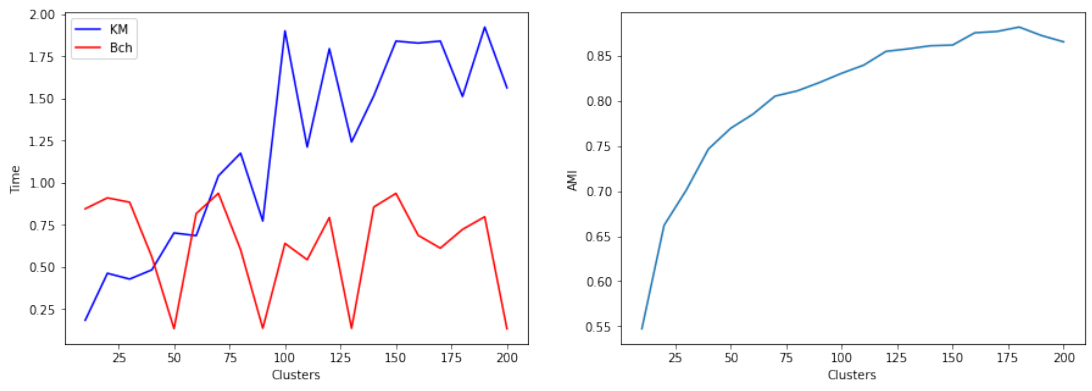


BIRCH

BIRCH is an algorithm that compresses the data to certain granularity using the Leader algorithm to reduce the data size and uses an indexing datastructure for fast assignment, the adjustment of its parameters is key for good results. Also assumes that the whole dataset does not fits in memory but the compressed data is a good approximation of the data densities

[4]:
```
ltimes = []
ltimeskm = []
agree = []
brlb = []
kmlb = []

for nc in range(10, nclusters, 10):
    birch = Birch(threshold=0.02, n_clusters=nc, branching_factor=100)
    itime = time.perf_counter()
    brlb.append(mbkm.fit_predict(citypos))
    etime = time.perf_counter()
    ltimes.append(etime-itime)

for nc in range(10, nclusters, 10):
    km = KMeans(n_clusters=nc, n_init=1)
    itime = time.perf_counter()
    kmlb.append(km.fit_predict(citypos))
    etime = time.perf_counter()
    ltimeskm.append(etime-itime)

for birchlabels, kmlabels in zip(brlb,kmlb):
    agree.append(adjusted_mutual_info_score(birchlabels, kmlabels))

fig = plt.figure(figsize=(15,5))
ax = fig.add_subplot(121)
```

```
plt.plot(range(10, nclusters, 10), ltimeskm, color='b', label='KM')
plt.plot(range(10, nclusters, 10), ltimes, color='r', label='Bch')
plt.xlabel('Clusters')
plt.ylabel('Time')
plt.legend()
ax = fig.add_subplot(122)
plt.plot(range(10, nclusters, 10), agree);
plt.xlabel('Clusters')
plt.ylabel('AMI');
```



The implementation of scikit-learn of BIRCH does not always do a good job because the final number of clusters is obtained using agglomerative hierarchical clustering

4.5.2 Increasing the size of the dataset

Now let's see what happens if we increase the size of the dataset. For this we are going to generate randomly spherical blobs of with increasing numbers of examples.

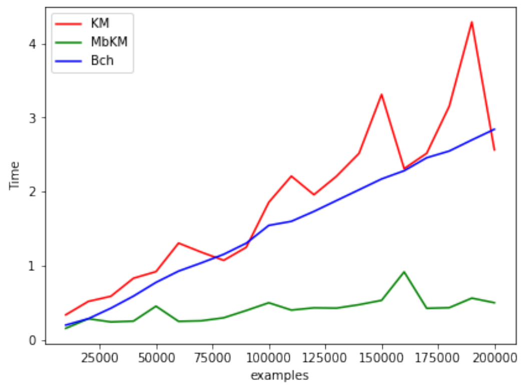
The three algorithms are computed for the data and the time used and the agreement with the original labels is collected. The size of the batches of minibatch K-means and Birch branching factor are set to a 5% of the size of the dataset, this is just a random guess and these parameter have to be carefully set.

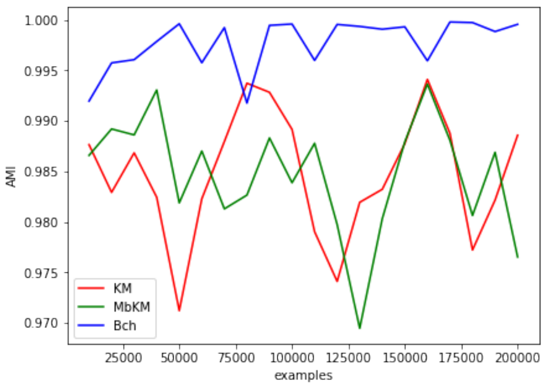
If you run this cell again it takes some time to complete. You have also this code in the /Code folder so yu can run it outside the notebooks and explore different options.

[5]:
```
ltimes = []
agree = []
step = 10000
limit = 200001
ncenters = 50
for ns in range(step, limit, step):
    blobs, labels = make_blobs(n_samples=ns, n_features=20,␣
 ↪centers=ncenters, cluster_std=1, center_box=(-3,3))
    # KMeans
    km = KMeans(n_clusters=ncenters, n_init=1)
    itime = time.perf_counter()
    kmlabels = km.fit_predict(blobs)
    etime = time.perf_counter()
    dtime1 = etime-itime
    agg1 = adjusted_mutual_info_score(labels, kmlabels)
    # Minibatch Kmeans
    itime = time.perf_counter()
    mbkm = MiniBatchKMeans(n_clusters=ncenters, batch_size=int(ns/20),␣
↪n_init=1)
    mbkmlabels = mbkm.fit_predict(blobs)
    etime = time.perf_counter()
    dtime2 = etime-itime
    agg2 = adjusted_mutual_info_score(labels, mbkmlabels)

    # Birch
    itime = time.perf_counter()
    birch = Birch(threshold=5, n_clusters=ncenters, branching_factor=int(ns/
↪20))
    birchlabels = birch.fit_predict(blobs)
    etime = time.perf_counter()
    dtime3 = etime-itime
    agg3 = adjusted_mutual_info_score(labels, birchlabels)

    ltimes.append((dtime1, dtime2, dtime3))
    agree.append((agg1, agg2, agg3))
```

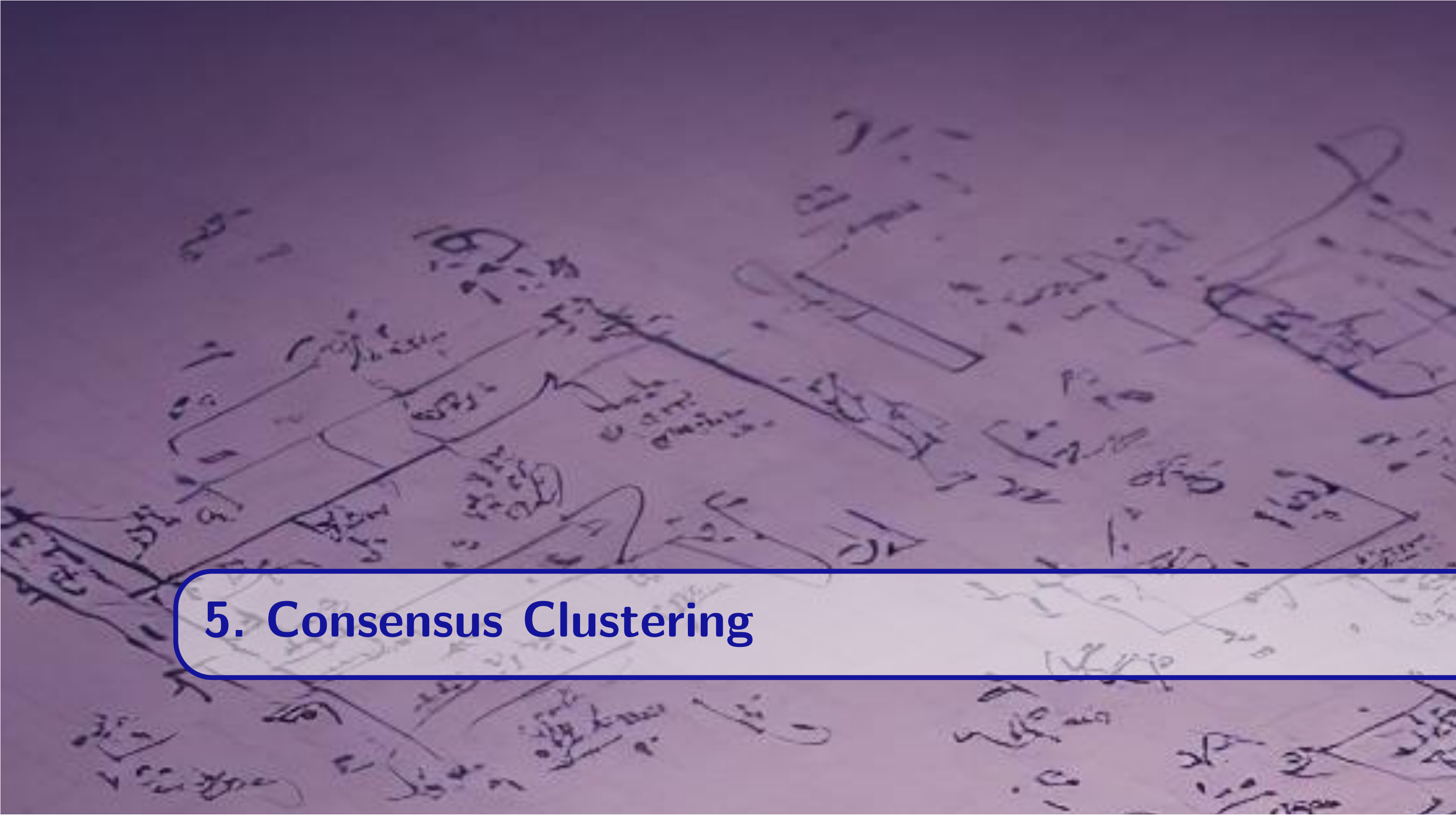
As you can see K-means time grows faster and Birch and K-means scale in different ways, be aware that changing their parameters also change the time cost. Looking to the agreement to the original partitions you can see that Birch does in general a better job.

```
[6]: fig = plt.figure(figsize=(15,5))
     ax = fig.add_subplot(121)
     plt.plot(range(step, limit, step), [x for x,_,_ in ltimes], color='r',␣
      ↪label='KM')
     plt.plot(range(step, limit, step), [x for _, x,_ in ltimes], color='g',␣
      ↪label='MbKM')
     plt.plot(range(step, limit, step), [x for _, _, x in ltimes], color='b',␣
      ↪label="Bch")
     plt.xlabel('examples')
     plt.ylabel('Time')
     plt.legend()
     ax = fig.add_subplot(122)
     plt.plot(range(step, limit, step), [x for x,_,_ in agree], color='r',␣
      ↪label='KM')
     plt.plot(range(step, limit, step), [x for _, x,_ in agree], color='g',␣
      ↪label='MbKM')
     plt.plot(range(step, limit, step), [x for _, _, x in agree], color='b',␣
      ↪label="Bch")
     plt.legend();
     plt.ylabel('AMI')
     plt.xlabel('examples');
```





You can play with the size of the datasets and specially with the number of clusters, you will see that with a small number of clusters if data fits in memory in general K-means is really fast.



5. Consensus Clustering

5.1 Cluster ensembles

The combination of groups of classifiers in order to improve accuracy is a well established strategy in supervised learning. This technique can also be applied to unsupervised learning. The assumption is that combining different partitions it is possible to obtain a consensuated partition that improves over the individual ones, thus the name of *consensus clustering*, also known as *clustering ensemble*.

The problem can be defined in the following way, given a set of partitions of the same data $\mathcal{X}$:

$$\mathbb{P} = \{P^1, P^2, ..., P^n\}$$

with:

$$\begin{aligned} P^1 &= \{C^1_1, C^1_2, ..., C^1_{k_1}\} \\ &\vdots \\ P^n &= \{C^n_1, C^n_2, ..., C^n_{k_n}\} \end{aligned}$$

where C^i_j is the j-th cluster of partition P^i and k_i is the number of clusters of partition P^i, and the sum of the cardinalities of the clusters of the partition is $|\mathcal{X}|$, the goal is to obtain a new partition that uses the information of all n partitions with the following characteristics:

- *Robustness*, the combination of the partitions has a better performance than each individual partition in some sense
- *Consistency*, the combination is similar to the individual partitions
- *Stability*, the resulting partition is less sensitive to outliers and noise

- *Novelty*, the combination is able to obtain partitions that can not be obtained by the clustering methods that generated the individual partitions

Consensus clustering also adds some advantages over classical clustering algorithms:

- *Knowledge reuse*, given that the consensus can be computed directly from the partition assignments, previous partitions from the data using the same or different attributes can be introduced in the process.
- *Distributed computing*, the individual partitions can be obtained independently, so the computational cost of obtaining the partitions to consensuate can be distributed in different processes.
- *Privacy*, only the assignments of the individual partitions are needed for the consensus, so partitions that use attributes with sensitive information do not need to be shared to obtain the final partition.

The different methods for consensus clusterings are based usually in a two step process, the first step is to generate the individual partitions to be combined, the second step is the process that combines the partitions to generate the final partition. The next sections will describe the different techniques used for these steps and will give some details of the main consensus clustering methods in the current literature.

5.2 Partition generation

One of the main ideas of supervised ensemble generation is that the key to a better performance resides in the quality and diversity of the individual classifiers to combine. This is also true for cluster ensembles. These are the main techniques used for generating the individual clusters:

- Different example representations: mainly the diversity is obtained by generating the partitions using different subsets of the attributes of the data. This allows for partitions that use different and complementary perspectives of the data similarly to the techniques used for example in random forest classifiers.
- Different clustering algorithms: to take advantage that all clustering algorithms have different biases and that can produce different partitions using the same data.
- Different parameter initialization: some clustering algorithms are able to produce different partitions using different parameters. This includes parameters that change directly or indirectly the number of clusters and the starting point of the algorithms based on an objective function that is optimized using local search (for instance k-means).
- Subspace projection: Using dimensionality reductions techniques like random projections or random cuts allow producing different clusterings from different perspectives.
- Subsets of examples: Based also on techniques used by supervised ensembles, random subsamples (bootstrapping) of the dataset allow generating diverse clusterings. More powerful supervised methods like boosting that allow re-weighting the samples are not available given that we are in the unsupervised domain.

There are some literature ([9, 44, 57, 99]) about how the diversity of the set of cluster affects the quality of the resulting consensus. The main conclusions are that when individual clusterings are too diverse this has a harmful effect on the final quality and a moderate diversity is more desirable. The means to assess the diversity of the individual partitions rely mainly on external validity indices like the ones used for clustering validation (a popular choice is the Normalized Mutual Information index (NMI)). These indices can be used as a distance among partitions and allow choosing the partitions that are on average not too distant from the others. There are also consensus techniques that use the measure of diversity to change the influence of each individual partition in the consensus partition.

Given a set of diverse clusters generated using any of the mentioned techniques, the next problem is how to combine them.

5.3 Consensus process

Once we have the individual partitions, we need to use the information of the assignments of the examples to determine the consensuated partition. We can divide the different techniques in two groups, those that are based on the objects co-ocurrence matrix and those that are based on the median partition problem.

The methods that use the co-cocurrence of examples as consensus information specifically the coincidence of pairs of examples in the same cluster through the different partitions. The idea is that if different partitions put together the same examples that means that these examples should be together in the consensus partition. The methods based on the mean partition problem assume that there is a partition that, given a similarity function among partitions, is equally similar to all the individual partition, so the goal is to search in the space of possible partitions to find this consensus partition.

5.3.1 Co-ocurrence based methods

These methods use in some way the labels obtained from each individual clustering and the coincidence of the labels for the examples in different clusterings. There are several ways to exploit this information, the main methods are based on relabeling and voting, the co-association matrix, graph and hyper-graph partitioning, information theory measures and finite mixture models.

Relabeling and voting

Methods based on relabeling and voting first have to solve the labeling correspondence problem. Each partition assigns a symbolic label to each example that is unrelated to the labels of other partitions. This is a non-trivial problem that is solved heuristically using bipartite matching algorithms and cumulative voting. This correspondence problem also only can be solved accurately if the number of clusters in the individual partitions is the same, restricting also the resulting partition to the same number of clusters as the input clusterings. The heuristic methods for solving the correspondence are also expensive (for example the Hungarian algorithm for bipartite matching is $O(n^3)$), this makes these methods not suitable when the number of clusters is large.

For instance, in [32] a voting strategy is performed incrementally. Each new clustering votes for the cluster that an example belongs to, and, after that voting, each example has the number of times it has been assigned to a specific cluster. Each new clustering has to be relabeled to determine the correspondence with the current consensus clustering, this problem is solved approximately with a heuristic procedure using the confusion matrix between the consensus and the new clustering or by using the Hungarian algorithm. The resulting clustering can be used as a fuzzy partition (the degree corresponds to the proportion of votes) or the maximum values can be used to obtain a hard partition.

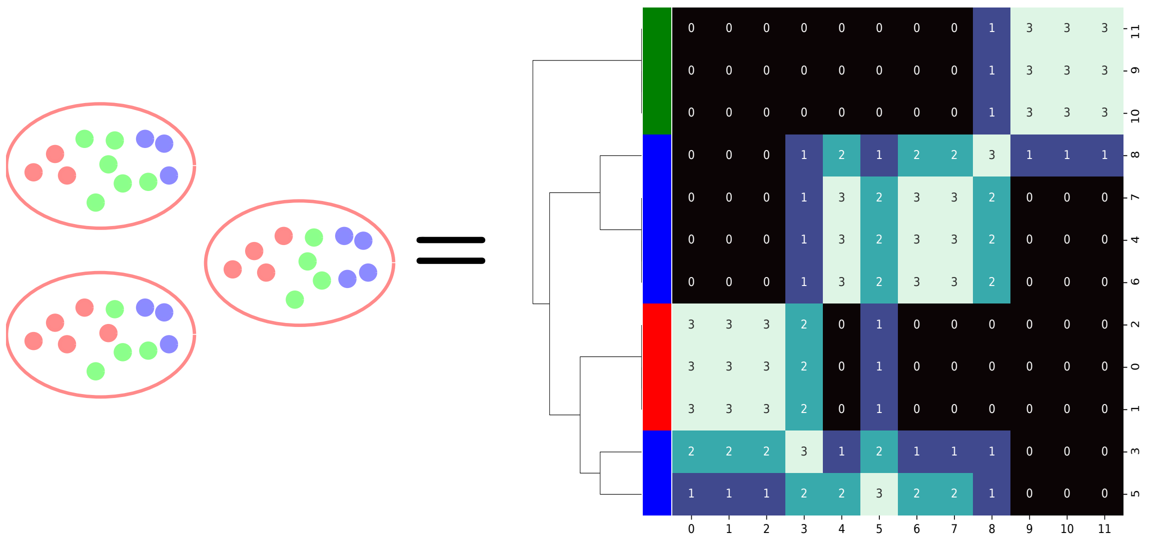


Figure 5.1: Consensus partition by clustering the co-association matrix

Co-Association matrix

Methods based on the co-association matrix avoid solving the correspondence problem by computing a square matrix that counts for all the input partitions how many times a pair of examples appear in the same cluster. This matrix is used as the input for the consensus function. This matrix can be used as a similarity function, considering that if a pair of examples has a larger count this means that are more similar. Applying a clustering algorithm to this matrix a consensuated clustering can be obtained. The main computational issue of all these methods is to compute the matrix, that is quadratic in time and space respect to the number of examples, so it is not suitable for large datasets.

For instance, in [46] the co-association matrix is thresholded to 0.5 and the connected components of the resulting graph are returned as the consensus clustering. In [47] the proposal is to apply to the association matrix different hierarchical clustering algorithms as single linkage, complete linkage, average linkage and using a heuristic to cut the dendrogram to obtain the final clustering. An example of this procedure is shown in figure 5.1.

The argument of [71] is that the information obtained from the coassociation matrix is not fine-grained enough and that augmenting the information using similarities inferred from the graph induced from this matrix can result in better consensus. The idea is to use a similar approach than the measures from link similarity like the Pagerank algorithm. Two measures based on triplets are defines. The weight of a link between two clusters is computed as the number of examples they have in common.

The distance called Weighted Connected Triple (WCT) is computed as:

$$WCT^z_{xy} = min(w_{xz}, w_{yz}) \; ; \; WCT_{xy} = \sum_{\forall z} WCT^z_{xy} \; ; \; S_{WCT}(C_x, C_y) = \frac{WCT^z_{xy}}{WCT^z_{max}} \times DC \tag{5.1}$$

with $DC \in [0,1]$ a decay factor.

The distance called Weighted Triple-Quality (WTC) is defined from the measure:

$$WTQ^z_{xy} = \frac{1}{\sum \forall v_t \in N_z w_{zt}} \tag{5.2}$$

where N_z is the set of vertices connected to z with a weight different from 0. The final distance is computed as the previous one. A combined distance that multiplies the previous distances is also proposed.

With the matrix computed with these distances for the clusters obtained for all the clusterings, a matrix for the instances and clusters is obtained. Each example is weighted for each cluster with the computed distance.

Using this matrix two consensus functions are proposed. The first one uses the new matrix as a new set of features and then a clustering algorithm is applied, in this case K-means and PAM. The use of bipartite graph partitioning is also proposed. In this case the spectral graph partitioning algorithm SPEC is used.

Graph and hypergraph partitioning

Graph and hypergraph methods transform consensus to a graph partitioning problem. The different methods differ on how the graph is obtained from the individual clusterings and the criteria used to cut the graph. The main advantage of these methods is that they are usually linear respect to the number of examples, this makes them more suitable for large datasets.

The first proposal for these algorithms is found in [126]. Three algorithms for consensus clustering are proposed, all based on transforming the problem to a hyper-graph. This transformation is performed in the following way:

For each individual clustering λ_i the binary membership indicator matrix H_i is computed with a column for each cluster. There are as many rows as objects in the cluster. The matrices from each clustering are concatenated and this defines the adjacency matrix of a hyper-graph with n vertices and $\sum_{\lambda \in \Lambda} k^{(\lambda)}$ hyper-edges.

The Cluster based Similarity Partitioning Algorithm (CSPA) is based on computing the similarity among clusters using the co-occurrence of objects in the same cluster. A consensuated similarity matrix can be computed using the averaged similarity among pairs of clustering. This similarity can be computed as:

$$S = \frac{1}{r} H H^T \tag{5.3}$$

The consensuated clustering is obtained by partitioning the similarity matrix using the METIS graph partitioning algorithm. The main problem of this approach is the quadratic space and time complexity in the number of objects.

For instance, given the following set of clusterings $\{C_1, C_2, C_3\}$ for a set of five examples:

	C_1	C_2	C_3
x_1	1	2	1
x_2	1	2	1
x_3	1	1	2
x_4	2	1	2
x_5	2	3	2

We can compute the indicator matrix H as follows:

	$C_{1,1}$	$C_{1,2}$	$C_{2,1}$	$C_{2,2}$	$C_{2,3}$	$C_{3,1}$	$C_{3,2}$
x_1	1	0	0	1	0	1	0
x_2	1	0	0	1	0	1	0
x_3	1	0	1	0	0	0	1
x_4	0	1	1	0	0	0	1
x_5	0	1	0	0	1	0	1

Computing the similarity S matrix we obtain:

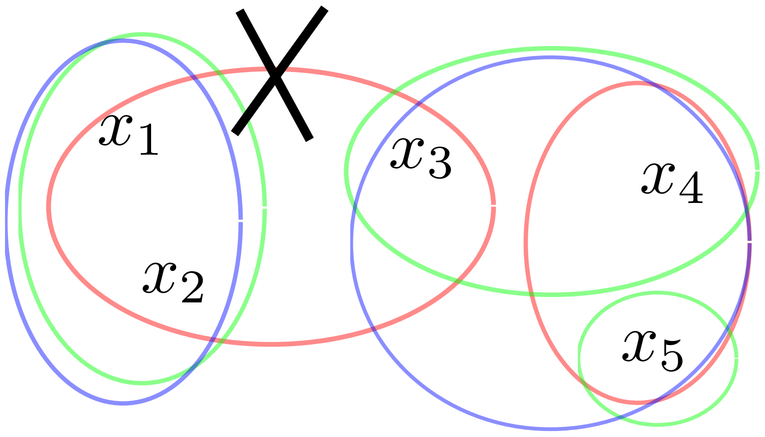


Figure 5.2: Partition of the hyper-graph of a set of clusterings using HPGA

$S =$

	x_1	x_2	x_3	x_4	x_5
x_1	1	1	1/3	0	0
x_2	1	1	1/3	0	0
x_3	1/3	1/3	1	2/3	1/3
x_4	0	0	2/3	1	2/3
x_5	0	0	1/3	2/3	1

In the HyperGraph-Partitioning Algorithm (HGPA) the problem is formulated as a partitioning of the hyper-graph from the original hyper-graph by cutting the minimal number of hyper-edges. The graph is partitioned into k unconnected components of approximately the same size (this avoids trivial partitions). It has to be noticed that if the data on the original clusters is unbalanced this is not a good approach. The HMETIS hyper-graph partitioning algorithm is used to partition the hyper-graph. The cost of this algorithm is $O(k \cdot n \cdot m)$. Figure 5.2 shows the hyper graph built from the clusterings of the previous examples and a partition of the graph to obtain two clusters.

In the Meta-CLustering Algorithm (MCLA) the idea is to group and collapse hyper-edges and assign the objects to the hyper-edge in which it participates the most. The first step is to construct a meta-graph, the hyper-edges are used as vertices of the meta graph, the weight associated with the edges among the vertices is calculated as the binary Jaccard coefficient between hyper-edges. As there is not overlapping among clusters of the same clustering the graph obtained is r-partite. The second step is to cluster the hyper-edges using the METIS algorithm. Each meta-cluster has approximately r vertices. This gives a correspondence among the labels of the clusterings. The number of cluster obtained is k. The third step is to collapse the meta-clusters, for each one of the k meta-cluster the hyper-edges are collapsed into a single meta hyperedge. Each hyper-edge has an association vector that contains an entry for each object and the level of association to the meta-cluster. The level is computed by averaging the values of the indicator vectors. The fourth step is to assign each object to the most associated meta-cluster. Not all meta-clusters are guaranteed to win at least one object. The cost of this algorithm is $O(k^2 \cdot n \cdot m^2)$

Figure 5.3 shows a meta-graph obtained from the clusterings of the previous example. An edge appears among two clusters if the value of the Jaccard coefficient between the columns corresponding to the clusters is different from 0. The weights assigned to each edge correspond to the value of the Jaccard coefficient. Applying the MCLA algorithm would partition the meta-graph into k clusters. The examples would be assigned depending on their affinity to the original clusters inside each cluster from the partitioning of the meta-graph.

An improvement over these methods is presented in [43]. The main criticism of previous

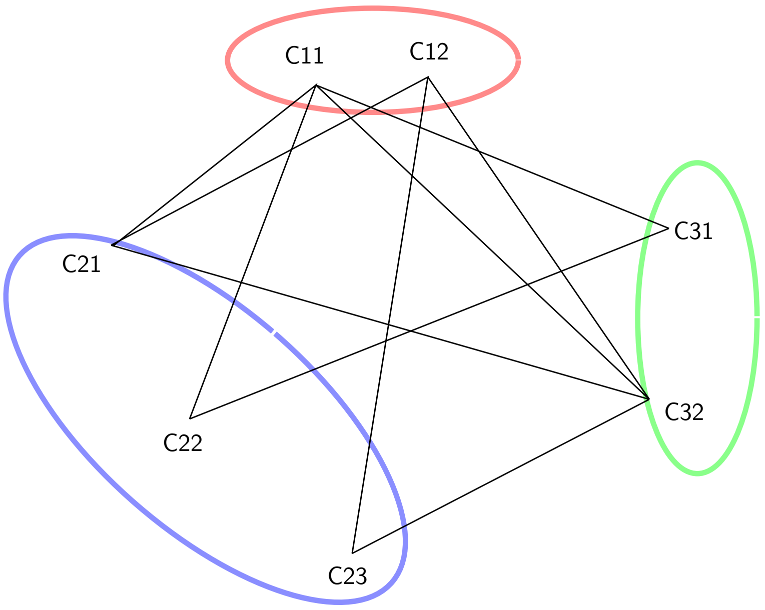


Figure 5.3: Metagraph for a set of clusterings for consensus with the MCLA algorithm

methods is not to consider the similarity of the instances and the clusters together as it is considered as a loss of important information. The Hybrid Bipartite Graph Formulation (HBGF) is proposed to correct that. This method constructs a graph $G = (V, W)$ with the individual partitions $C = \{C^1, C^2, ..., C^R\}$ where:

- $V = V^C \cup V^I$ where V^C contains t vertices, each one representing a cluster of the ensemble and V^I contains n vertices representing the instances of the dataset
- W is defined in the following way, if the vertices are both clusters or instances its weight is 0 if instance i belong to cluster j is 1, otherwise is 0

This representation allows reconstructing the individual clusterings and uses the combined similarities of instances and clusters. The graph is partitioned using either spectral graph partitioning or the METIS algorithm. The cost of this algorithm is $O(k \cdot n \cdot m)$.

Information Theory

Information Theory based methods use measures from information theory to assess partition similarity. An example is the methods proposed in [128]. Their approach relates the problem of consensus to the problem of finding the median partition (the best partition that summarizes a set of partitions), but the consensus is solved using the labels co-association.

The median partition is derived from the similarity among partitions. There are different functions that can be used to compare partitions. For example the category utility from COBWEB.

$$U(\pi_C, \pi_i) = \sum_{r=1}^{K} p(C_r) \sum_{j=1}^{K(i)} p(L_j^i | C_r)^2 - \sum_{j=1}^{K(i)} p(L_j^i)^2 \tag{5.4}$$

Where $p(C_r) = |C_r|/N$, $p(L_j^i) = |L_j^i/n$ and $p(L_j^i|C_r) = |L_j^i \cap C_r|/|C_r|$.

This measure can be rewritten as the Goodman-Kruskal index for the contingency table of two partitions. The overall utility of a partition with respects of all the partitions can be measured as the sum of individual utilities:

$$U(\pi_C,\Pi) = \sum_{i=1}^{H} U(\pi_C,\pi_i) \tag{5.5}$$

So the best partition is the one that maximizes this measure.

It is proved that the maximization of partition utility is equivalent to minimization of the square-error clustering criterion if the number of classes is fixed. The labels of the clusterings can be transformed to a new set of features with continuous values

$$\overline{y}_{ij}(x) = \delta(L_j^i,\pi_i(x)) - p(L_j^i), forj = 1,...,K(i), i = 1,...,H \tag{5.6}$$

With this transformation the median partition can be calculated using the K-means algorithm. This makes this method more related to the co-occurrence of examples. The number of clusters of the consensus partition has to be determined a priori. The computational cost of the algorithm is $O(knm)$.

Finite Mixture Models

The methods based on Finite Mixture Models transform the problem to the estimation of the probability of the assignment of the examples to the cluster labels. For instance, also in [128] a consensus function is proposed based on the transformation of the attributes of the dataset so the consensus can be solved as a problem to maximum likelihood estimation. This transformation uses the labels of the different clusters as new attributes, so the original attributes are discarded, and the consensus is obtained by the clustering of the data in this new space. The premise is that the new dataset is drawn from a mixture of multivariate distributions, one for each clustering. So the data can be modeled as:

$$P(y_i|\Theta) = \sum_{m=1}^{M} \alpha_m P_m(y_i|\theta_m) \tag{5.7}$$

Assuming that the data is independent an identically distributed the log likelihood function for the parameters $\Theta = \{\alpha_1,...,\alpha_M,\theta_1,...,\theta_M\}$ given the dataset Y is:

$$logL(\Theta|Y) = log\prod_{i=1}^{N} P(y_i|\Theta) = \sum_{i=1}^{N} log \sum_{m=1}^{M} \alpha_m P_m(y_i|\theta_m) \tag{5.8}$$

The consensus can be obtained by maximizing the log likelihood.

To model the problem, the probability density function is modeled as a multinomial trial:

$$P_m^{(j)}(y_{ij}|\theta_m^{(j)}) = \prod_{k=1}^{K(j)} \vartheta_{jm}(k)^{\delta(y_{ij},k)} \tag{5.9}$$

The consensus can be performed using the Expectation Maximization algorithm. The number of clusters of the consensus partition has to be fixed a priori to determine the number of components of the mixture model. This method has the advantage that it can be used in the case of datasets with missing values. This is a common scenario if the clusters are obtained by sampling from the original dataset. The computational cost of the algorithm is $O(knm)$.

5.3.2 Median partition based methods

These methods compute the consensus solving the median partition problem. This problem can be stated as: given a set of partitions $\mathcal{P}$ and a similarity function among partitions $\Gamma(P_i,P_j)$, to find the partition P_c in the partition space of the dataset $\mathbb{P}_x$ that maximizes the similarity to the set:

$$P_c = \arg\max_{P \in \mathbb{P}_x} \sum_{P_i \in \mathcal{P}} \Gamma(P, P_i) \tag{5.10}$$

This is a more theoretically founded method and there have been studies of the computational cost of finding that partition for some distances. For example, for the Mirkin distance it has been proven that the problem is NP-hard, but there are no results for other distances.

There are several similarity functions that can be used to compute the similarity among partitions, most of them are based on external cluster validity indices, for instance:

- Similarities based on counting the agreements and disagreements of pairs of examples between two partitions like the Rand index, the Jaccard coefficient or the Mirkin distance (and their randomness adjusted versions).
- Similarities based on set matching can be also used, like the Purity or the F-measure.
- Similarities based on information theory measures (compute how much information two partitions share) as the NMI, the Variation of Information and the V-measure.

There are some methods proposed for solving the mean partition problem using the Mirkin distance. This distance computes the sum of disagreements between two partitions (pairs of examples that are clustered together by one partition and separately by the other). Solving the problem exactly is computational unfeasible for large datasets (NP-hard) so some heuristics are proposed. For instance, the *Best of k* heuristic chooses from the set of partitions the one that minimizes its distance to all the other partitions, this has the drawback that the resulting partition is one of the original partitions.

Other heuristics approach the problem as an optimization problem using local search meta-heuristics. The simplest one applies a hill climbing strategy that starts from a partition and each iteration chooses the best movement of one example from its cluster to another, considering the best movement as the one that increases the most the similarity from the consensus partition to the rest. The main drawback is the dependence on the initial partition and the problem of falling on poor local minimum solutions.

In the same line, with the aim to avoid local minimum, simulated annealing can be used as meta-heuristics. An initial partition is generated (possibly at random) and one random movements of an example from one cluster to another is generated each iteration. The movement can be accepted or not depending on the annealing schedule function.

Genetic algorithms have also been proposed as multicriteria optimization methods to solve jointly the consensus problem and the selection of the number of clusters. Each individual encodes the assignment of the examples to clusters and different operators for crossover and mutation can be employed.

A different approach to solve the problem is to use Non-negative Matrix Factorization (NMF). In [86] a consensus clustering is defined as an optimization problem that can be reduced to the problem of clustering the association matrix among partitions.

Being $\tilde{M}$ the average association matrix of a set of partitions, the problem is defined as:

$$min_{ij} \sum_{i,j}^{n} (\tilde{M}_{ij} - U)^2 \tag{5.11}$$

This can be reformulated as a NMF problem. The matrix U has to enforce a set of constraints that have to hold the consensus partition, namely transitive relations among the instances (i

belongs to the same cluster as j and j to the same cluster as k then i and k belong to the same clusters) a cubic number of these constraints have to be solved to obtain the consensus partition.

These same constraints can be solved factorizing the matrix U as a product HH^T being H a $n \times k$ matrix with values $[0,1]$ with only a 1 at each row. This problem is relaxed and transformed into a problem that has to optimize:

$$min_{ij} \sum_{i,j}^{n} (M - QSQ^T)^2 \tag{5.12}$$

with Q and S non negative matrices and $Q^TQ = I$. The matrix Q contains the assignments of the examples.

5.4 Notebook: Consensus Clustering

```
In [1]: from sklearn.datasets import load_iris,  make_moons, make_circles
        from sklearn.metrics import adjusted_mutual_info_score
        import matplotlib.pyplot as plt
        import seaborn as sns
        from kemlglearn.datasets import make_blobs
        from sklearn.cluster import KMeans
        from kemlglearn.cluster.consensus import SimpleConsensusClustering
        import numpy as np
        from numpy.random import normal

        import warnings
        warnings.filterwarnings('ignore')

        data = load_iris()['data']
        labels = load_iris()['target']
```

The `kemlglearn` library has an implementation of a simple consensus algorithm based on the coassociation matrix.

The basis classifier is K-means, it has the following parameters:

- `n_clusters` = Number of clusters
- `n_clusters_base` = Number of clusters to use the base classifier
- `n_components` = Number of components of the consensus
- `ncb_rand` = If the number of clusters of each component is chosen randomly in the interval [2..`n_clusters`]

We will start applying consensus clustering to the iris dataset and we will compare with a single k-means.

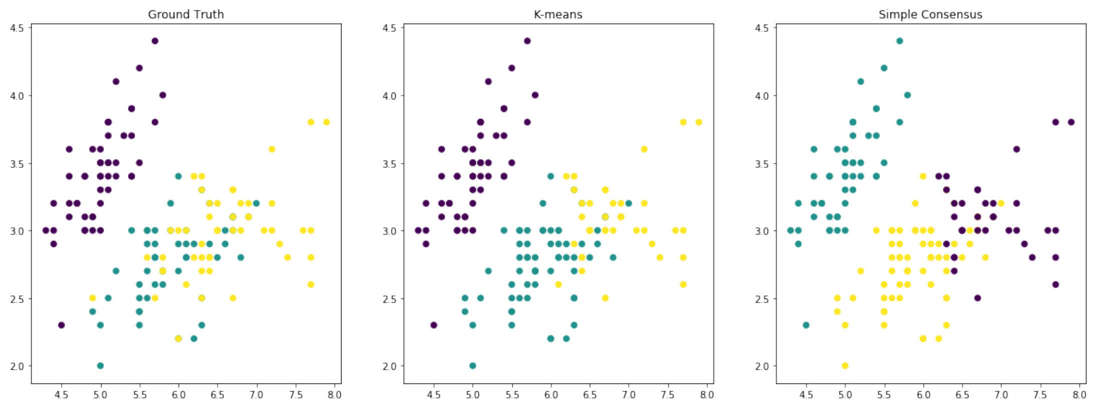
Feel free to experiment with the parameters of the consensus to see if there is any improvement respect to the default values for the parameters.

We will start with the iris dataset, in this case there is a slighly improvement for the AMI respect to the ground truth using the consensus by combining 30 clusterings with 10 clusters each. Obviously in a real application we will not have the ground truth and we will need to explore the hyperparameters of the consensus clustering method, but we can use internal quality methods to assess the quality of the clustering.

```
In [2]: nc = 3
        km = KMeans(n_clusters=nc)

        cons = SimpleConsensusClustering(n_clusters=nc, n_clusters_base=10,␣
 ↪n_components=30, ncb_rand=False)

        lkm = km.fit_predict(data)
        cons.fit(data)
        lcons = cons.labels_

        print('K-M AMI =', adjusted_mutual_info_score(labels, lkm))
        print('SCC AMI =', adjusted_mutual_info_score(labels, lcons))
```

```
K-M AMI = 0.7551191675800484
SCC AMI = 0.7954205025674187
```

```
In [3]: fig = plt.figure(figsize=(20,7))
        ax = fig.add_subplot(131)
        plt.scatter(data[:,0],data[:,1],c=labels)
        plt.title('Ground Truth')
        ax = fig.add_subplot(132)
        plt.scatter(data[:,0],data[:,1],c=lkm)
        plt.title('K-means')
        ax = fig.add_subplot(133)
        plt.scatter(data[:,0],data[:,1],c=lcons)
        plt.title('Simple Consensus');
```



Now we will apply the consensus clustering to two clusters of different sizes and densities where K-means usually has difficulties. We can experiment with the parameters of the consensus but also with the characteristics of the dataset, in this case we will see what happens with a particular dataset and parameters.

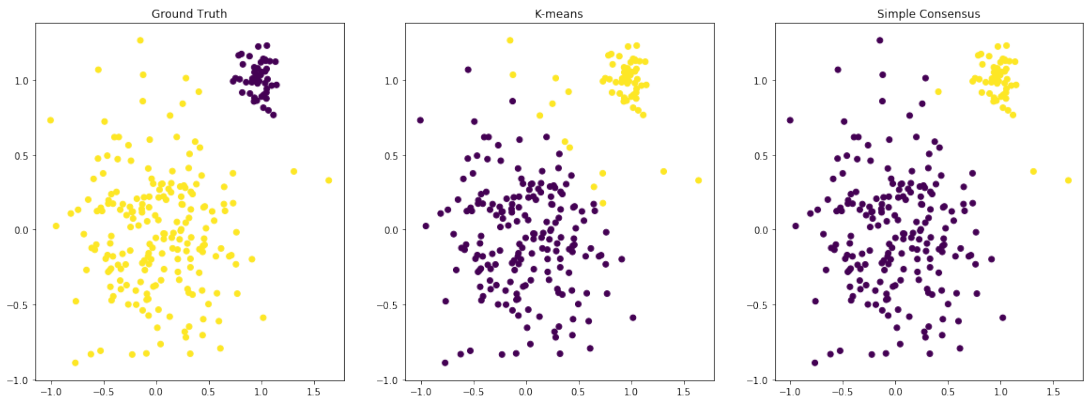
```
In [4]: data, labels = make_blobs(n_samples=[50, 200], n_features=2,␣
 ↪centers=[[1,1], [0,0]], random_state=2, cluster_std=[0.1, 0.4])
```

```
In [5]: nc = 2
        km = KMeans(n_clusters=nc)
        cons = SimpleConsensusClustering(n_clusters=nc, n_clusters_base=20,␣
 ↪n_components=50, ncb_rand=False)

        lkm = km.fit_predict(data)
        cons.fit(data)
        lcons = cons.labels_

        print('K-M AMI =', adjusted_mutual_info_score(labels, lkm))
        print('SCC AMI  =', adjusted_mutual_info_score(labels, lcons))
```

```
K-M AMI = 0.6977028093054155
SCC AMI  = 0.8929632461306622
```

We can see a clear difference on the quality of the consensus clustering, notice that the base clustering have 20 clusters each. Usually a large nunber of clusters has advantages because we group the data at a certain granularity and mixing all the partitions allows to discover more clearly how data is distributed.

```
In [6]: fig = plt.figure(figsize=(20,7))
        ax = fig.add_subplot(131)
        plt.scatter(data[:,0],data[:,1],c=labels)
        plt.title('Ground Truth')
        ax = fig.add_subplot(132)
        plt.scatter(data[:,0],data[:,1],c=lkm)
        plt.title('K-means')
        ax = fig.add_subplot(133)
        plt.scatter(data[:,0],data[:,1],c=lcons)
        plt.title('Simple Consensus');
```



Now we will apply consensus clustering to elongated clusters that are also difficult for K-means

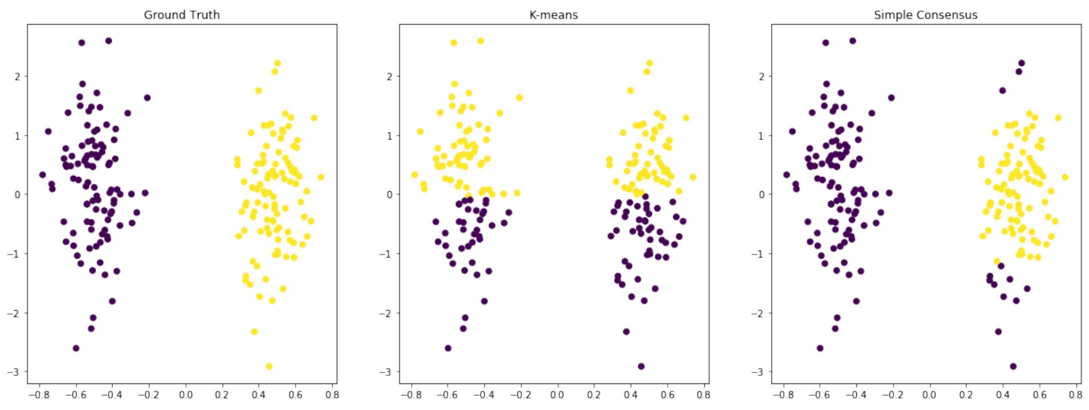
```
In [7]: sc1=100
        v1=0.1
        sc2=100
        v2=0.9

        data = np.zeros((sc1+sc2,2))
        data[0:sc1, 0] = normal(loc=-0.5, scale=v1, size=sc1)
        data[0:sc1, 1] = normal(loc=0.0, scale=v2, size=sc1)
        data[sc1:, 0] = normal(loc=0.5, scale=v1, size=sc2)
        data[sc1:, 1] = normal(loc=0.0, scale=v2, size=sc2)
        labels = np.zeros(sc1+sc2)
        labels[sc1:] = 1
```

```
In [8]: nc = 2
        km = KMeans(n_clusters=nc)

        cons = SimpleConsensusClustering(n_clusters=nc, n_clusters_base=20,␣
 ↪n_components=50, ncb_rand=False)

        lkm = km.fit_predict(data)
        cons.fit(data)
        lcons = cons.labels_


        print('K-M AMI =', adjusted_mutual_info_score(labels, lkm))
        print('SCC AMI  =', adjusted_mutual_info_score(labels, lcons))
```

```
K-M AMI = 0.0023795995094841966
SCC AMI  = 0.7123645529599809
```

The results will not always be good, but it will be more consistent than K-means. It will require some parameter experimentation though.

```
In [9]: fig = plt.figure(figsize=(20,7))
        ax = fig.add_subplot(131)
        plt.scatter(data[:,0],data[:,1],c=labels)
        plt.title('Ground Truth')
        ax = fig.add_subplot(132)
        plt.scatter(data[:,0],data[:,1],c=lkm)
        plt.title('K-means')
        ax = fig.add_subplot(133)
        plt.scatter(data[:,0],data[:,1],c=lcons)
        plt.title('Simple Consensus');
```



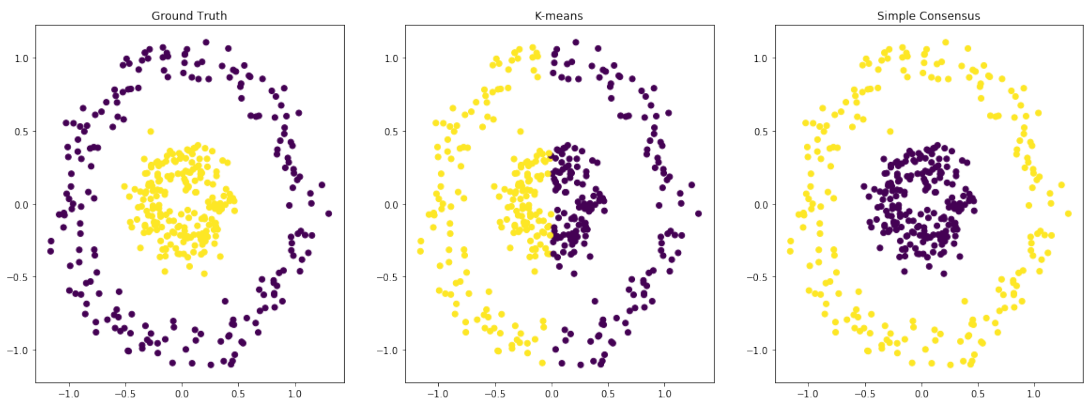
This is the two rings dataset where K-means can not generate the true clusters.

```
In [10]: data, labels = make_circles(n_samples=400, noise=0.1, random_state=4,␣
 ↪factor=0.3)

In [15]: nc = 2
         km = KMeans(n_clusters=nc)

         cons = SimpleConsensusClustering(n_clusters=nc, n_clusters_base=20,␣
 ↪n_components=50, ncb_rand=False)

         lkm = km.fit_predict(data)
         cons.fit(data)
         lcons = cons.labels_

         print('K-M AMI =', adjusted_mutual_info_score(labels, lkm))
         print('SCC AMI  =', adjusted_mutual_info_score(labels, lcons))
```

```
K-M AMI = -0.0013617681496602037
SCC AMI  = 0.9772422365106967
```

Consensus clustering is able to obtain almost the true clusters

```
In [16]: fig = plt.figure(figsize=(20,7))
         ax = fig.add_subplot(131)
         plt.scatter(data[:,0],data[:,1],c=labels)
         plt.title('Ground Truth')
         ax = fig.add_subplot(132)
         plt.scatter(data[:,0],data[:,1],c=lkm)
         plt.title('K-means')
         ax = fig.add_subplot(133)
         plt.scatter(data[:,0],data[:,1],c=lcons)
         plt.title('Simple Consensus');
```



This is the two moons dataset, it is more dificult than the previous ones.

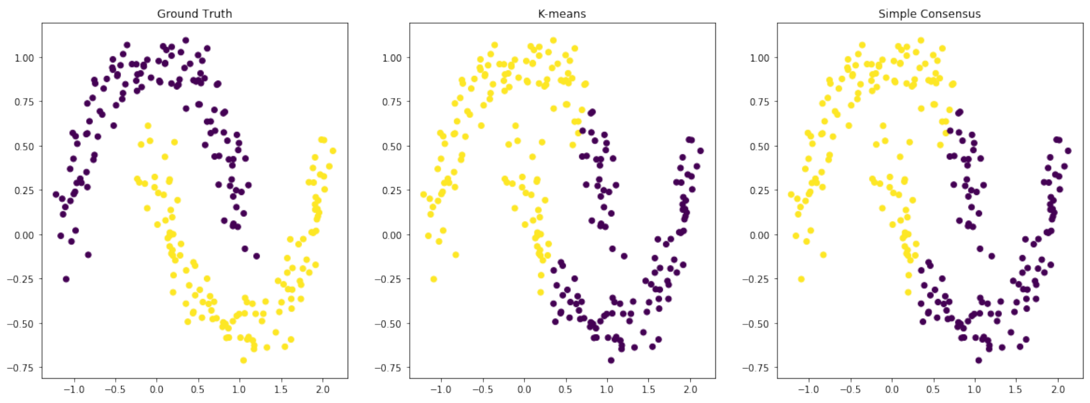
Consensus cluster is not silver bullet, on this case it is difficult to find good parameters that separate well both classes.

```
In [17]: data, labels = make_moons(n_samples=250, noise=0.1)
```

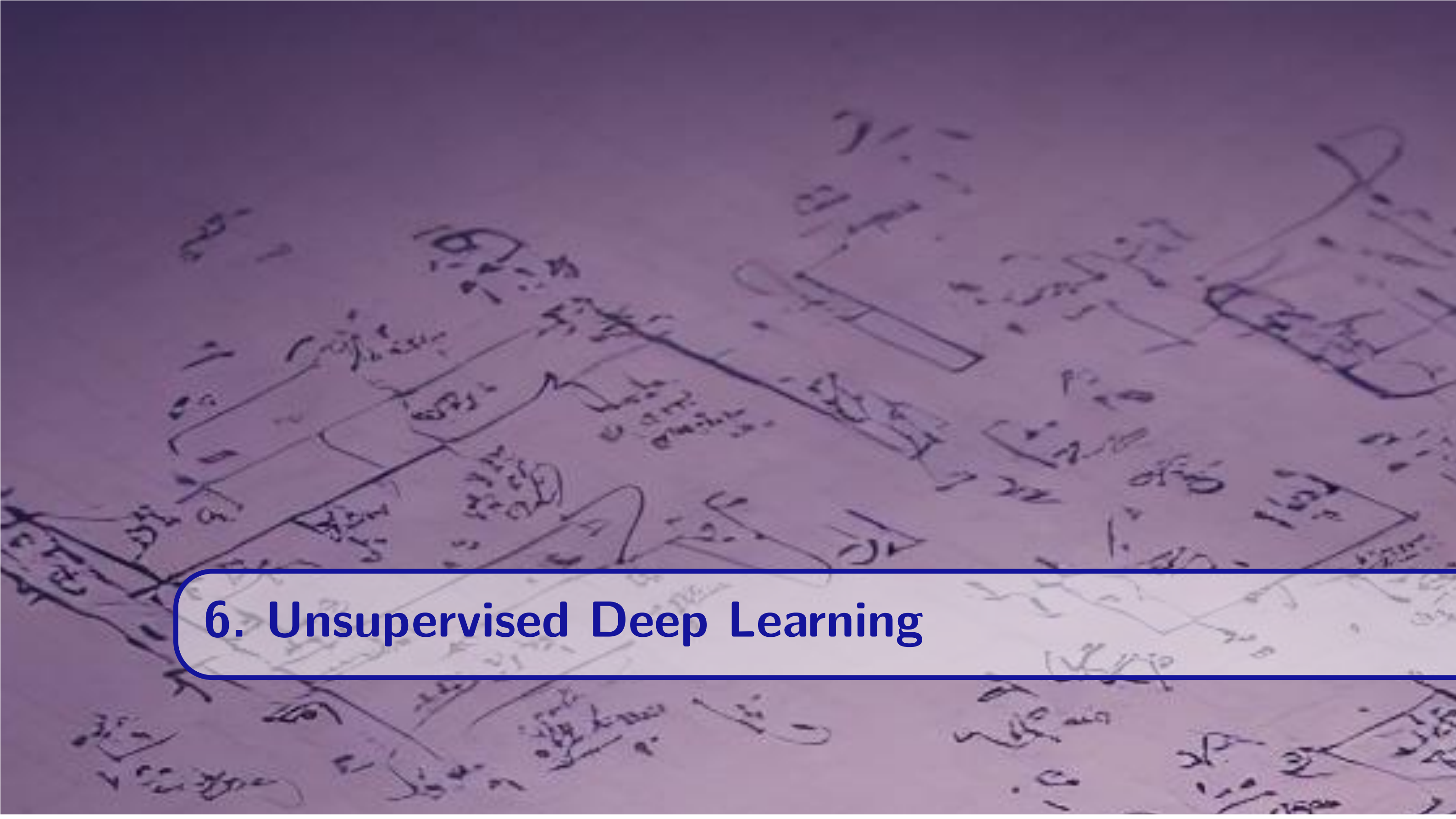
```
In [18]: nc = 2
         km = KMeans(n_clusters=nc)

         cons = SimpleConsensusClustering(n_clusters=nc, n_clusters_base=15,␣
 ↪n_components=150, ncb_rand=False)

         lkm = km.fit_predict(data)
         cons.fit(data)
         lcons = cons.labels_

         print('K-M AMI =', adjusted_mutual_info_score(labels, lkm))
         print('SCC AMI =', adjusted_mutual_info_score(labels, lcons))
```

```
K-M AMI = 0.18987762345390774
SCC AMI = 0.18987762345390774
```

```
In [19]: fig = plt.figure(figsize=(20,7))
         ax = fig.add_subplot(131)
         plt.scatter(data[:,0],data[:,1],c=labels)
         plt.title('Ground Truth')
         ax = fig.add_subplot(132)
         plt.scatter(data[:,0],data[:,1],c=lkm)
         plt.title('K-means')
         ax = fig.add_subplot(133)
         plt.scatter(data[:,0],data[:,1],c=lcons)
         plt.title('Simple Consensus');
```



III Unsupervised Deep Learning



6. Unsupervised Deep Learning

6.1 Introduction

As it happens in the conventional areas of machine learning, the areas of neural networks and deep learning are more focused on supervised tasks, mainly classification or regression. For instance many problems in machine vision solved using deep learning like image classification, object localization or image segmentation correspond to one of these two tasks. All these task need labels.

Considering that the datasets used in these tasks are very large, to be able to solve a specific problem supposes a very expensive journey until a model is able to perform reasonably well in the problem. It is also a problem that each task needs not only a specific dataset but also a specific set of labels that depends on the problem. The same images can be used for image classification or object localization, but the way the labels have to be defined is very different.

There are other tasks that do not involve labels or that can alleviate the need for labels. For instance, we can learn generative models that capture the statistical properties of our data that can be then used to generate novel examples with specific characteristics. We can also use unlabeled data (we have plenty of these) to solve unsupervised tasks that help to learn representations that capture the semantic structure of the data and that can then be used in for solving other supervised tasks via transfer learning acting as a bootstrapping representation or a task agnostic transfer learning method.

6.2 A dream waiting to happen

A few years before the beginning of the deep learning era (around 2006) there were different techniques that allowed training deep networks in a unsupervised way. For instance Deep Belief Networks were trained layer by layer using an auto encoder task. Each layer was trained and frozen to allow the training of successive layers. This obtained a set of initial weights that allowed training the network for a supervised task more efficiently.

This jump-started the work on deeper networks jointly with the use of specialized hardware (GPUs) that accelerated the training of even deeper networks. The success of training directly

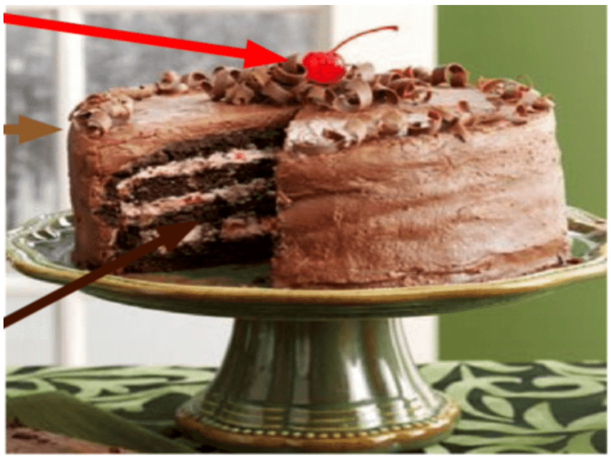


Figure 6.1: The famous Yann LeCun cake aka LeCake

supervised tasks with larger datasets jointly with an increase on computational power lead to a reduced interest in unsupervised training. Nevertheless, the idea of being able to train without labels has been always in the mind of the top researchers in the area.

In 2014, during one of the periodical AMA (ask me anything) meetings about Deep Learning from the Reddit artificial intelligence and machine learning communities, Geoffrey Hinton (one of the founders of the modern neural network research) commented:

> ”The brain has about 10^{14} synapses, and we only live for about 10^9 seconds. So we have a lot more parameters than data. This motivates the idea that we must do a lot of unsupervised learning since the perceptual input (including proprioception) is the only place we can get 10^5 dimensions of constraint per second.”

In several conferences, Yann LeCun (other of the top researches in neural networks) talking about the future of unsupervised deep learning has made famous his cake metaphor about how much information is able to use/predict each paradigm of machine learning. Reinforcement learning only predicts a reward signal from time to time, that accounts for a few bits of information, and corresponds to the cherry on top of the cake. Supervised learning predicts a label or a few numbers, this puts the scale on tens or thousands of bits, that is the icing of the cake. Unsupervised learning should be able to predict any part of its input from any observed part (for instance future frames of a video), that puts the scale in millions of bits, that takes the rest of the cake. According to him “The revolution will not be supervised”.

One of the researchers that has been the lead of unsupervised training in machine vision, Alexei Efros made a bet in 2014 after the success of the R-CNN architecture for object localization on the PASCAL VOC benchmark. The success of the network come from a pretraining using the Imagenet dataset. The bet was that by the next year, a network pretrained unsupervisedly would improve over that result. This is now known as the gelatto bet[1]. Unfortunately, he lost the bet, but things are getting close and it will probably happen in a few years. Something that makes people confident of this is that currently all networks that solve natural language tasks are based on BERT or GPT, that are trained this way.

[1]http://people.eecs.berkeley.edu/~efros/gelato_bet.html

6.3 Many methods, one goal

The area of unsupervised deep learning is not new, but as the supervised counterpart the use of GPUs has accelerated its development. The advantage is that unsupervised datasets are easier to build and can be order of magnitudes larger.

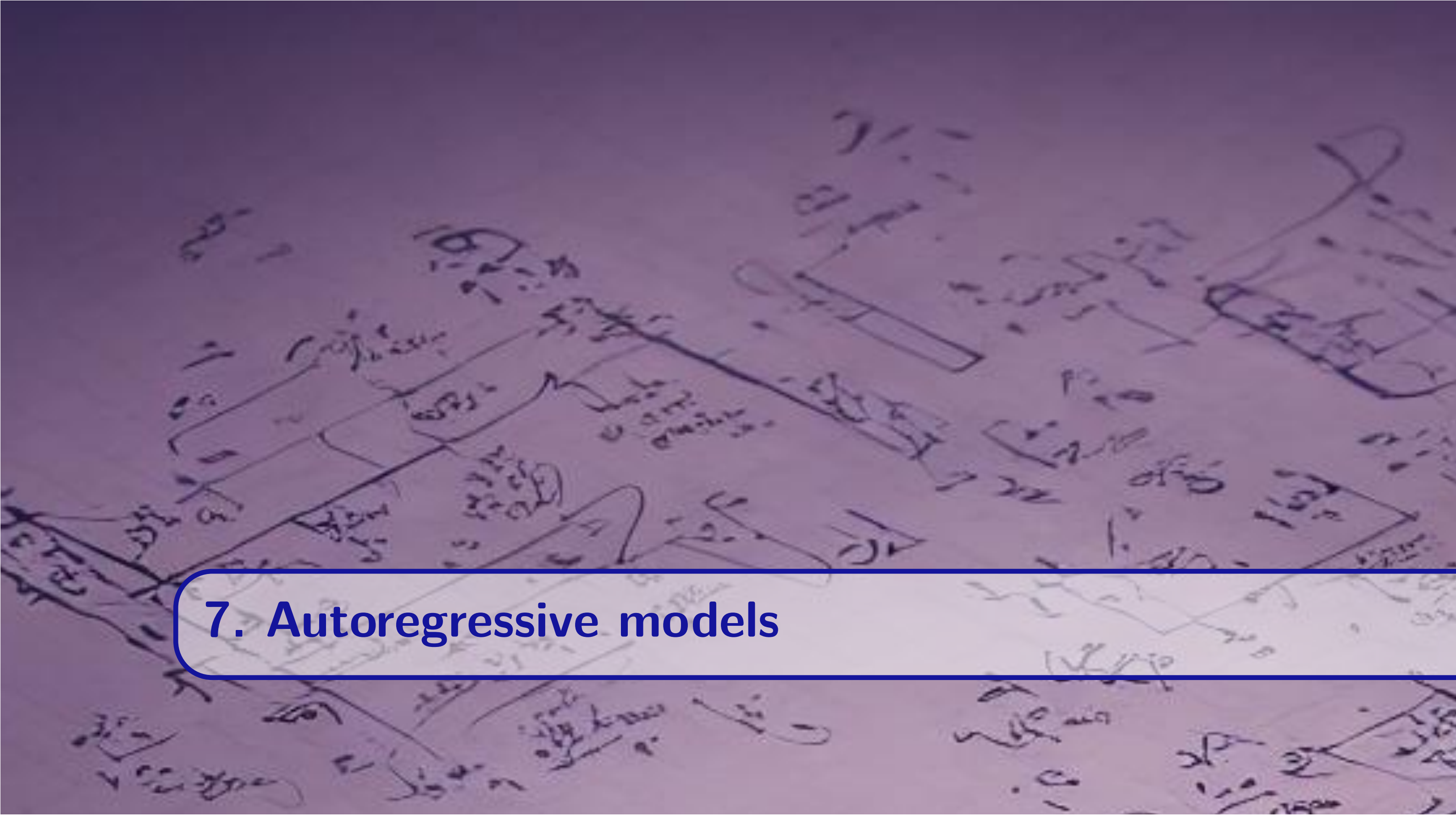
In the following chapters we are going to introduce several unsupervised deep learning methods that can be divided in two groups: generative models and self-supervised learning.

The objective of generative models is to capture the statistical distribution of data. The usual statistical distributions that we are accustomed to using for describing data are not enough to capture the complexities and dependencies of complex problems. Also, the high dimensionality of the data poses a huge problem for these distributions. Deep neural networks will be the basis of how the distributions will be learned, looking for representations able to capture the elements that describe the data.

There are several approaches that can be taken for learning probability distributions. We can take a parametric approach, assuming certain constraints in the form of the distribution or in the dependencies betweeh variables, where a neural network learns these parameters, or we can model the distribution non-parametrically, using the network for representing implicitly the data. There are different approaches that have their advantages and drawbacks, we are going to talk about these:

- Autoregressive methods, where the product rule of probabilities is used to transform the data distribution estimation into a sequence learning problem.
- Normalizing flows, where the probability distribution of the data is transformed to a known probability distribution (gaussian, uniform) through a reversible function that allows go back and forth between the distributions.
- Latent variable models, where the probability distribution is approximated using a known probability distribution in a latent space of lower dimensionality using an auto encoding network.
- Implicit models, where the distribution of the data is not represented explicitly, and the network is used to transform data from a known distribution in a latent space to samples from the data distribution trained using an adversarial game.
- Diffusion models, that uses a noising/denoising process in analogy to physical diffusion processes (for instance the expansion of gasses) that given certain constraints allows reversing the process to obtain the initial conditions from a final state (literally reversing time).

The aim of self-supervised learning is to train a network using a pretext task. The assumption is that general features can be learned from that task by a network that can be used for other supervised tasks by transfer learning or fine-tuning. We will focus mainly on self-supervised methods for computer vision. We will cover first some successful techniques to obtain good features without labels solving different pretext tasks like learning to colorize a gray scale image or to solve jigsaw puzzles. The lecture notes will end with methods included in the area of contrastive learning, that train networks to separate images in the feature space using distance learning methods.



7. Autoregressive models

7.1 Probabilistic models

All the generative models we are going to talk about learn one way or another a probability distribution that corresponds to the process that generates the data. Probability distributions are interesting because we can use them for many purposes. For instance, defining the specific types of variables that compose the distributions, what kind of base distributions they follow and what kind of relationship/influence they have, we can generate complex data like images, video, text, sound, speech...

Probability distributions also compress the information of the data that is captured in the parameters of the joint distribution and can be used as a learned representation. We can also perform different inferences that are useful for deciding if new data corresponds to the distribution, how typical they are or if an example belongs or not to the distribution (anomalous data/out of distribution).

There are different ways of learning probability distributions, but we are going to focus on distributions that can be estimated using **maximum likelihood**. We only need to define the base distribution that we assume the data is following and a dataset of examples of the process we want to model.

7.2 Maximum likelihood estimation

We are going to assume that you already know about maximum likelihood estimation from previous machine learning courses, so we will just briefly explain the basic ideas.

The approach of maximum likelihood is similar to other approaches like, for instance, empirical risk minimization, that is used by non-probabilistic machine learning models. The goal is to approximate $p_{data}(x)$ using a parameterized function $p_\theta(x)$. Basically, we are going to optimize a loss for finding a set of parameters that corresponds to the adjustment of the data to the probability distribution. We will learn the parameters θ using an optimization procedure so $p_\theta(x) \approx p_{data}(x)$. If the function is expressive enough, and we have plenty of data the estimation is guaranteed to converge to the actual parameters that generate the data.

In this particular case, we are using the likelihood of the data according to the corresponding probability distribution. In order to avoid numerical problems we use the logarithm of the likelihood, and we are going to assume that examples are independent and identically distributed, so we have this minimization problem:

$$\arg\min_\theta loss(\theta, x_1, \ldots, x_n) = -\frac{1}{n}\sum_{i=1}^{n} \log p_\theta(x_i)$$

In the end, this procedure is equivalent to minimizing the Kullback-Leibler divergence (KL) between the empirical data distribution and the model. We can proof that this is the case by developing the KL formula:

$$KL(P|Q) = \sum_i P(i)\log\frac{P(i)}{Q(i)} = \sum_i P(i)(\log P(i) - \log Q(i)) = \sum_i P(i)\log P(i) + \sum_i -\log Q(i)P(i)$$
$$= -H(P) + \mathbb{E}_{x\sim P}[-\log Q(x)]$$

where $H(P)$ is a constant for a given dataset, so the only term to optimize corresponds to the expectancy of the negative log likelihood over P, that is the data we use for the estimation.

As in other machine learning models, we can solve this optimization problem using stochastic gradient descent (SGD). This has the advantage of being reasonably scalable to large datasets. We can use neural networks as the function that approximates the probability distribution.

7.3 Autoregressive models

Not any architecture of neural networks will be able to represent an arbitrary probability distribution. It needs to represent a proper distribution, so the sum of the probabilities computed by the network for all the domain on the variables of the problem has sum one and also any probability that is computed has to be positive or zero. We need also the log of the distribution to be easy to compute and also to be a differentiable function respect to the parameters of the distribution, given that that is the key of the optimization with SDG.

Bayesian networks is a formalism that is used to represent joint probability distributions. They are directed acyclic graphs (DAG) that define the dependence/independence relationships among the variables of the distribution. Each node is representing $P(X_i|parents(X_i))$ where X_i is a variable from the data and $parents(X_i)$ are all the variables (nodes in the DAG) that are connected to that variable. Given a node of a bayesian network, we can represent its probabilities as a neural network. The joint probability of the distribution of the variables will be the product of all these probabilities represented by the neural networks. We can use these probabilities to compute the log likelihood:

$$\log p_\theta(x) = \sum_{i=1}^{d} \log p_\theta(x_i|parents(x_i))$$

We do not know what are the dependencies among the variables for an arbitrary dataset, but the product rule of probabilities allows us to write any joint probability distribution in the following way, given a set of variables $X_1, X_2, \ldots, X_n$:

$$p(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} p(X_i|X_1, X_2, \ldots, X_{i-1}) = \prod_{i=1}^{n} p(X_n|X_{<i})$$

This allows to define what are called **autoregressive models** that can write their log likelihood as:

$$logp(x) = \sum_{i=1}^{d} \log p(x_i|x_{<i})$$

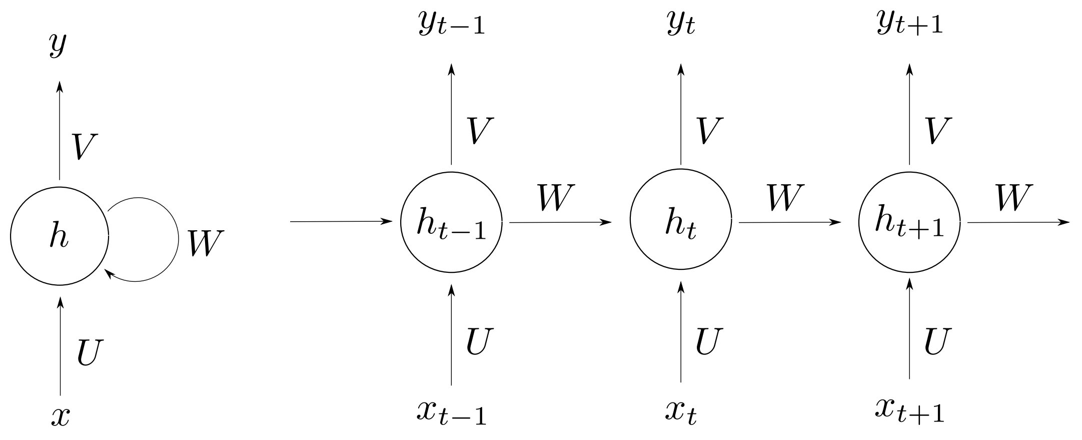


Figure 7.1: RNN node and RNN unrolled computation

Each variable corresponds to a node in the bayesian network that is connected to the precedent variables. We can use a neural network for predicting the value of each node and simply multiply their output to obtain the probability assigned by the joint probability distribution (or for computing the log likelihood).

Obviously it would be very inefficient to have as many neural networks as variables in our problem, so we can share parameters among all the networks helping this way also to the generalization.

There are two approaches that can be used for representing an autoregressive model with neural networks, using recurrent neural networks or using masking combined with different architectures like the MLP, convolutional networks or architectures based on attention.

7.3.1 Recurrent neural networks

Recurrent neural networks (RNN) are feed forward networks that are used for sequential data. In these networks, the variables for each example are introduced one by one, and there is a virtual connection between the computation of one step and the previous one.

As can be seen on the left of figure 7.1, each neuron has a state (h) that stores the combined computations performed to all the elements of the sequence until the current one. A recurrent node has an input (x) that corresponds to one element of the sequence, this connection can have linear transformation (U) that obtains a value that is combined with the state (h) using also a linear transformation (W). The output is obtained applying a non linearity after possibly another linear transformation (V).

This computation is performed for all the elements of the sequence one by one, represented by the computational graph on the right of figure 7.1. This is called the unrolled recurrent network. The weight matrices U, V and W are shared by all the computations. The computation stored on h at a specific point of the sequence corresponds to:

$$h^{(t+1)} = f(h^{(t)}, x_{t+1}; \theta) = f(f(h^{(t-1)}, x_t; \theta), x_{t+1}; \theta) = \cdots$$

We can see that this corresponds to an autoregressive model where for each step we could compute the probabilities of a variable using the value of all previous variables.

This model has several drawbacks that reduce its utility for representing probability distributions. The computations can not be done in parallel, given that each computation depends on the previous ones. This makes this model computationally expensive for long sequences. Other problem is that training presents exploding and vanishing gradients. This is due to the shared weight matrix that is multiplied over and over through the sequence and that, depending on their values for long sequences, could end on almost zero values if the values of the matrix are less than one or on a numerical overflow if the values are larger than one.

There are more advanced recurrent networks that reduce the problem of the exploding and vanishing gradients like Long Short Term Memory networks (LSTM) or Gated Recurrent Units (GRU). These are based on storing additional information on the recurrent cell and adding skip and gated connections that allow to control the flow of the gradient, helping to maintain it for longer sequences. These benefits are obtained unfortunately increasing the computational cost of each iteration that has to be performed to process the elements of the sequence. The additional computations can not be parallelized either.

Despite these drawbacks LSTMs and GRUs have been the state of the art in applications for natural language processing for a long time until the introduction of the transformer architecture. Recurrent networks are used in that area to represent the joint probability of words. The network can be trained with corpus of sentences or other sequences derived from NLP tasks like part of speech tagging or entity name recognition.

There have been different successful models based on recurrent network used for the generation of all kinds of sequential data. For instance, generation of literary text, computer code or web pages.

The generation of sequences is a natural task for recurrent networks given their autoregressive nature. To train a network it is only needed a dataset that corresponds to sequences composed by a vocabulary or set of tokens. Each element of the sequence is conditioned to the previous ones. A softmax is used as the output of the network acting as a classifier that computes probabilities for each possible token of the sequence. This output is interpreted as the probability of the next token corresponding to the part processed from the sequence.

Once the recurrent network is trained, the sampling new elements can be obtained by introducing an initial seed, obtaining the prediction for all the possible token symbols, choosing the most probable token, discarding the first token of the sequence and adding the new one, and feeding the new sequence to the network to obtain the next one. Usually there is a special token that corresponds to the end of sequence that appears on all the training sequences. The generation ends when that symbol is generated. This is a greedy method for sequence generation and does not guarantee the more probable sequence. An alternative is to perform a beam search where different alternative paths are maintained, choosing in the end the one that has the higher joint probability.

For instance, this is a random poem generated by a four layer LSTM trained with English poetry from the XVIII-XIX centuries. The training was done with sequences of characters and the network predicts the probability of the next character, so the network has no information about words or syntax. The sequence is generated autoregressively adding the most probable character to the sequence. Even when the text is basically meaningless (probably there are poetical styles where this kind of text will not be dissonant :-)), it could be appreciated some structure in the generation, like the length and punctuation of the verses, some attempts of rhyme (begun-run, fair-hair, sky-pray) and a mild respect for syntactical structure.

the sun roared and begun,
but line of war, among the sky,
and a sound is in his eyes;
and the true heard the sets so fair
and water thrickes hill,
o’er dark adore with his hull dream,
with a sheet for earth’s luck and run,
the roof with the border hair;

and the grey house put fled in a bird
the moon and out and sold;
there is no sight nor pray
the book with a silent weat;

7.3.2 Masked autorregressive networks

In order to reduce the computational cost of training the model an alternative is **masking**. The basic idea is to forbid the computation of connections in the network when these imply the interaction between variables that do not correspond to the autoregressive order.

It is called masking because the way to do this is to introduce in the computations matrices composed by 0/1 values that make zero the computations that are not allowed without affecting the ones that follow the correct order. This is implemented this way given that the training of neural networks is performed using specialized hardware (GPU/TPU) that is optimized for matrix multiplication, and can not deal with sparse matrices. To perform the computations using sparse matrices would not be efficient because we would not be able to take advantage of the parallelization that is provided by the hardware.

We will discuss solutions based on two approaches, masked MLP architectures and masked Convolutional architectures.

Masked MLPs

MADE (Masked Autoencoder for Distribution Estimation) [50] is an auto encoder network that estimates the joint probability distribution for sequences of binary/multinomial variables. In this case the input corresponds to the sequence of binary/multinomial variables to estimate, and each output corresponds to their probability given all the previous variables in the sequence obtained as usual with a sigmoid or a softmax.

The autoregressive order is obtained by introducing for each weight matrix a mask matrix that controls what connections are really active, since an MLP is fully connected. The mask matrices will have a pattern of 0s and 1s defined so each variable x_d only has connections from the variables $x_{<d}$. There is no constraint on the number of neurons that a hidden layer can have, so one variable could be connected to others by multiple paths as long as the autoregressive order is respected.

For instance, in figure 7.2 we have an autoencoder with the corresponding masks that deactivate all connections that do not correspond to the autorregressive order assuming that the first variable in the order is x_2, then x_3 and finally x_1. The variable x_1 does not have any connections forward given that it has no influence on the rest of variables. The output of variable x_2 given that it is the first one on the chosen order, has no connections and will correspond to the prior probability of that variable after training.

For any architecture with an arbitrary number of layers and units per layer, what connections are allowed between the hidden layers is defined by a set of rules. First, a number between 1 and $D-1$ is assigned to each unit of the input and output layer, that corresponds to the variable it is representing. Each neuron in each hidden layer will have a value $m^l(k)$ that is a number between 1 and $D-1$, that means the maximum value that can have an input neuron that has a path that connects to it. Each layer will have a mask matrix (M^{W^l}) that will set to 1 the autoregressive connections defined by these rules and the rest will be 0. For the neurons of the output layer there will be connections only with neurons with values lower than the number assigned to it defined by the mask matrix M^V. Following these rules the input layer will have a neuron that will not be connected to anything and the output layer will have a neuron that will not receive any connection

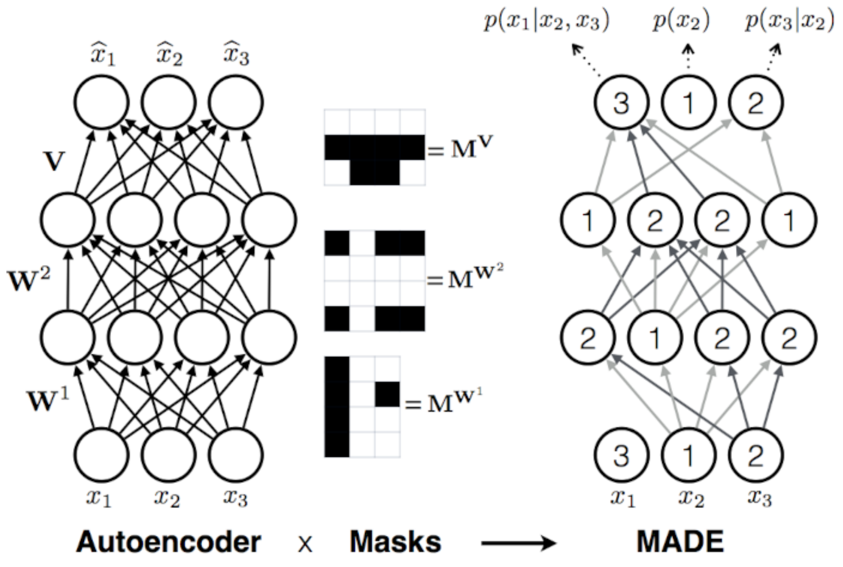

$\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$
V
W²
W¹
x_1 x_2 x_3
= M^V
= M^{W²}
= M^{W¹}
$p(x_1|x_2,x_3)$ $p(x_2)$ $p(x_3|x_2)$
Autoencoder
x
Masks
MADE

Figure 7.2: MADE (Masked Autoencoder for Distribution Estimation)[50]

In the hidden layers the computation performed is:

$$h^l(x) = g(b + (W^l \odot M^{W^l})x)$$

For the output layer the computation performed is:

$$\hat{x} = \sigma(c + (V \odot M^V)h^L(x))$$

To obtain better results and to improve the generalization of the network, several tricks are used during training, like using different input orders for the variables and using different masks that correspond to different autorregressive orders for the layers. This can be performed in parallel. An additional trick is to ensemble different models and averaging their results.

Inference in this model has to follow also the autoregressive order, so first the independent variable has to be sampled from the learned distribution at its corresponding neuron $p(x_1)$. This sampled value, is used as the value for the first input of the network (the rest of inputs can be randomly assigned), that will be used to compute the distribution for the second variable, since it only has connections to the first variable we are computing $p(x_2|x_1)$. We sample this new distribution that will give us the second input, and we repeat this procedure until completing the sequence. The last step will give us the complete sample. As it can be observed, the main handicap of this method is the computational cost of the inference. It needs to run the network as many times as variables has the problem and at each iteration all the weights of the network are used..

Masked 1D Temporal Convolutions

The main problem of MLPs is the number of connections that we have in the network. Even if they are masked, we need to have them in our parameters to be able to use parallel computations. An alternative is to use convolutions that allow reducing the number of connections given that they only focus on locality. This will reduce the computational cost given the reduction on the number of connections. However, he cost of the inference will remain, given that it has to follow the autoregressive order.

We can define convolutions on any number of dimensions, given that we are processing sequences we can define one dimensional convolutions with kernels that are $[1,k]$, so the

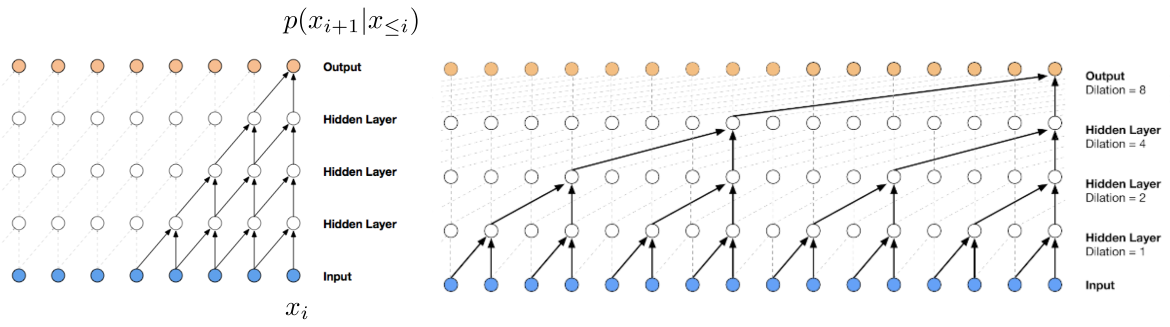


Figure 7.3: 1D causal convolutions and causal dilated convolutionsfor autoregressive models [107]

convolution is applied to only k elements in the sequence. Given that we have an order between the elements of the sequence we can not use the ordinary convolution that applies the kernel on the central element. In this case the convolution will be over the last element using the $k-1$ previous elements, so no information from the future of the sequence is used in the computation. This can be interpreted as a sort of masking, in this case we are masking the future. This kind of convolution is called *causal convolution*.

As we can use many layers on our network, the effective receptive field of the convolutions will increase with the layers as we will process the results of the previous layers. As a way of increasing this field without the need to increase much the number of layers it also can be used dilated convolutions. Dilated convolutions extend the convolution kernel processing longer sequences by only processing 1 of each n elements of the sequence, using n as dilation. For instance, in a sequence `ABCDEF`, a normal convolution with kernel of size 2 would process the pairs `[AB, BC, CD, DE, EF]`, but the same kernel with dilation 2 would process the pairs `[AC, BD, CE, DF]`.

Wavenet [107] is a model that uses these convolutions for voice generation from raw audio, in this domain sequences are very long. To increase further the receptive field of the convolutions, the dilation on each layer is increased exponentially. This increases the receptive field also exponentially. This model also uses other tricks to be able to generate audio, like quantizing the signal and predicting a discrete probability distribution and using skip and residual connections as in other convolutional architectures for reducing the problem of vanishing gradients. You can see the difference between causal and dilated causal convolutions on figure 7.3. Wavenet is the model that has been the basis of Google assistant. A blog entry describing the model and some demos of the audio that can be generated using this network can be found here.

Masked Spatial Convolutions

If we were to apply one dimensional convolutions to images we would lose a lot of information that could be useful for modeling. We can extend the idea of masking to two-dimensional convolutions with the adequate masking and using a specific order for the autorregresive distribution. This order is usually the raster order (top-bottom, left-right), but other orders are possible since this order is basically arbitrary.

This method is used by the model PixelCNN [130]. A mask that respects the raster order is used, so the convolution only uses the precedent pixels on that order as is represented in figure 7.4. As in Wavenet, several layers using these convolutions are stacked for increasing the receptive field and residual connections among layers are used to help generalization. As in the other models the prediction is autoregressive. This time we have a $n \times n$ image with three channels, each channel for each pixel has to be predicted using the precedent sampled values (it follows raster order and for each pixel samples R, G and then B). Each output is a softmax of

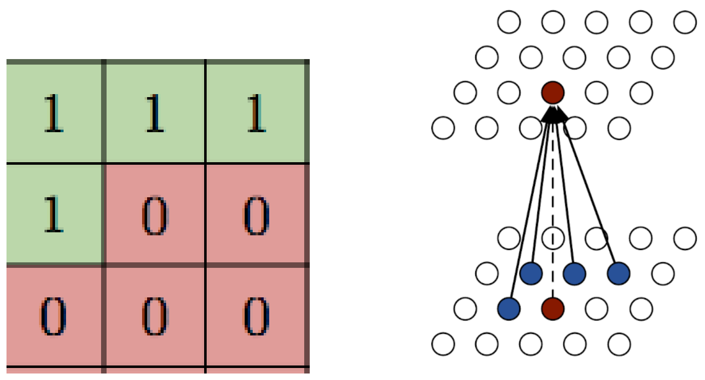


Figure 7.4: 2D masked convolutions following the raster order [130]

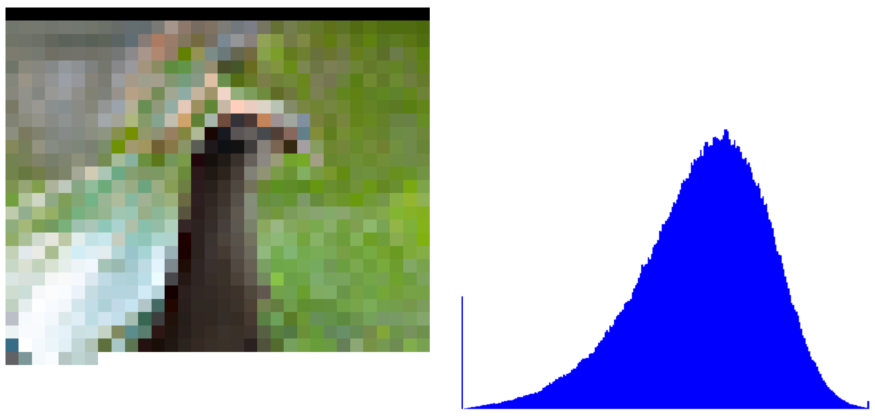


Figure 7.5: Inference on PixelCNN following the autorregressive order [130]

size 256, the possible intensities of the channel (see figure 7.5).

The use of 2D masked convolutions has a problem that propagates as we add new layers because of the way the convolution is masked. A blind spot appears in the receptive field that reduces the information that can be used. This is represented on the left-hand side of the figure 7.6. This is fixed in the sequel to this method called Gated Pixel CNN [106]. As can be seen on the right-hand side of the figure 7.6, two kinds of convolutions are used, a 1D causal convolution that process all the pixels on the same row as the current pixel and a 2D masked convolution that only uses the pixels above the current pixel and masking the rest. This combination of two kinds of convolutions allows increasing the receptive field without losing information.

The architecture of this network can be seen in figure 7.7. A layer is composed by two branches that process the two kinds of convolutions. Each layer use a gated architecture (hence the name of Gated Pixel CNN) where half of the computed feature maps are passed through *sigmoid* activations (the output is in the range [0,1]) and determine the effect of the other half that is passed through *tanh* activations (the output is in the range [-1,1]). This has the effect of fuzzy digital gates where the sigmoid activations act as a continuous gate that determines the effect of another branch of the network. The *tanh* determines if the effect is going to be positive or negative. Each branch has this gated architecture, the feature maps computed by the 2D convolutions are combined with the ones from the 1D convolutions using a 1×1 convolution branch that adapts the dimensionality of both sets of feature maps. The feature maps from the 1D convolutions that come from the previous layer are also passed by a residual connection for

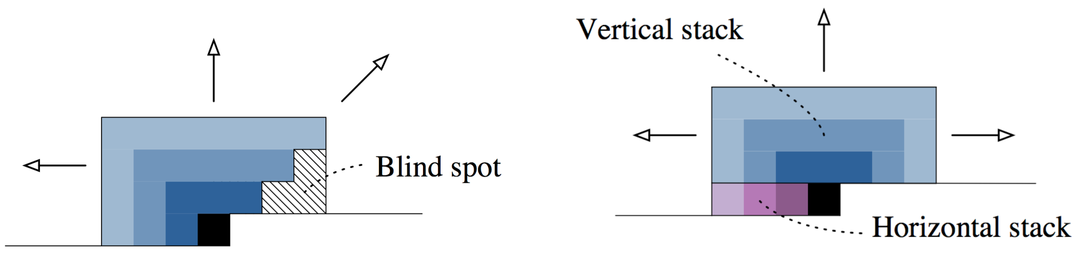


Figure 7.6: 2D masked convolutions generate a blind spot as new layers are added[106]

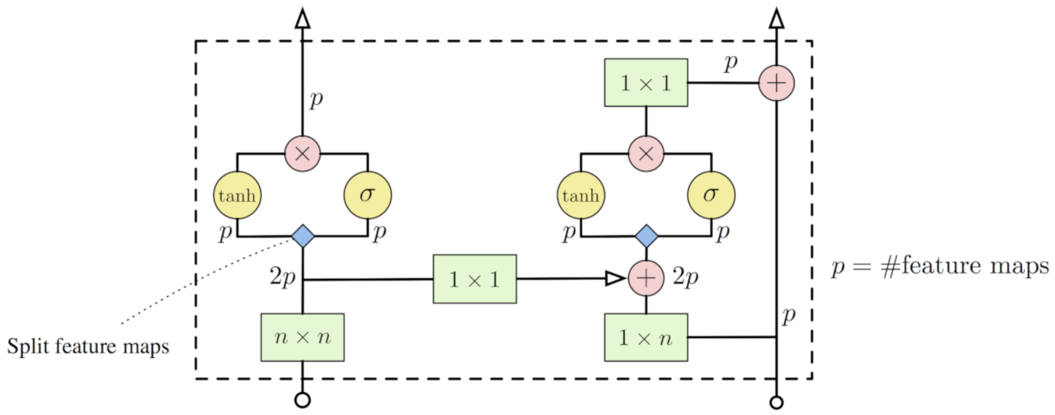


Figure 7.7: Architecture of Pixel CNN[106]

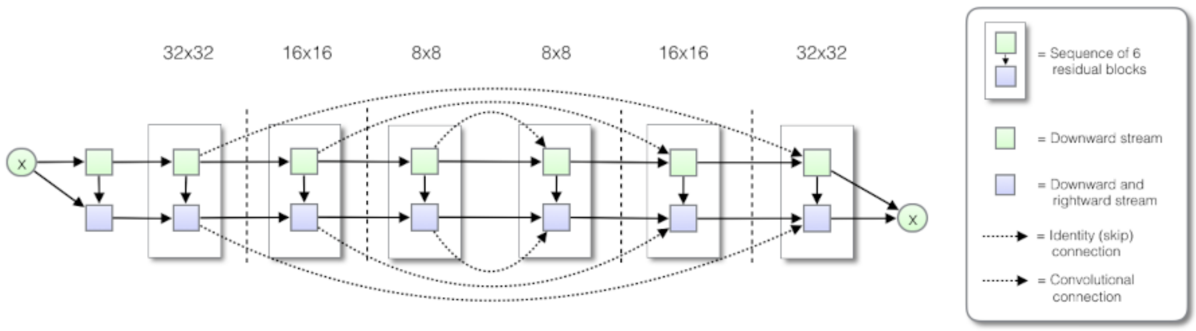

32x32
16x16
8x8
8x8
16x16
32x32
x
x
= Sequence of 6 residual blocks
= Downward stream
= Downward and rightward stream
= Identity (skip) connection
= Convolutional connection

Figure 7.8: Architecture of Pixel CNN++[119]

helping the flow of the gradient. In the end each layer computes two sets of feature maps of cardinality p that are passed to the final softmax.

The use of a softmax for computing the output is problematic, since it needs a large amount of memory, and it is not aware of the order of the pixels. We wan to model a discretized continuous value instead of a multinomial distributed value that is what the softmax is representing. Also being able to use the order of the pixels can help to generalization. Pixel CNN++ [119] is an improvement over this network that changes the softmax by a continuous approximation based on the logistic distribution. A mixture of k logistic distributions approximate the quantized pixel space, with a continuous distribution defined by:

$$p(x|\pi,\mu,s) = \sum_{i=1}^{K} \pi_i[\sigma((x+0.5-\mu_i)/s_i) - \sigma((x-0.5-\mu_i)/s_i)]$$

with the constraint that $\sum \pi_i = 1$. This reduces the number of parameters to store to only $2k$, where k will be much smaller than the number of pixel values. This value can be fixed a priori. A little trick that is used for training this distribution with discrete values is, instead of using directly the pixel values, to dequantize the values by adding uniform distributed noise to the pixels. A random value is obtained from the uniform $\mathcal{U}(-0,5,0.5)$ and is added to each pixel simulating a continuous distribution. On inference time the values are quantized again by rounding their values.

Some additional modifications are also introduced to the original architecture shown in figure 7.8. The network is transformed into a U-Net with a downsampling and a upsampling of the input and additional skip connections that allow the flow of information between layers that correspond to the same scale.

Masked Attention

Using convolutions we have a limitation on the length of the previous elements that we can take in account and hence the dependencies that are capture by a model. A more suitable solution is *Self Attention* [131] that has no limitation on the receptive field (all the sequence is used), the number of parameters does not depend on the dimensionality and all computations can be done in parallel.

The attention mechanism is the key element of the transformer architecture that has been popularized by the applications of deep learning to Natural Language Processing. It is faster than recurrent networks since it processes a sequence in parallel. It computes what is the influence/importance of the elements of an input sequence on an output sequence and uses this computation to modify the values of the input sequence according to these computed importances. We are not going in detail about how the transformer architecture works, but we are going to summarize the essential elements of Self Attention.

There are different procedures for computing attention, the Self Attention mechanism basically assumes that the input and output sequences are the same one. There are defined three inputs for attention, the query (Q), the key (K) and the value (V). The query and the key are combined using vector outer product obtaining a matrix. This defines the interactions between the elements of the sequence. This matrix can be masked to forbid the computation of certain interactions. In our case, a matrix can be used for instance to make zero all product that do not follow the autoregressive order, but there is more flexibility on what kind of masking can be applied. This matrix is scaled and a softmax is used to compute a set of weights that are multiplied to the value. The computation can be summarized as:

$$Attention(Q,K,V) = softmax(\frac{QK^\top}{\sqrt{d_k}})V$$

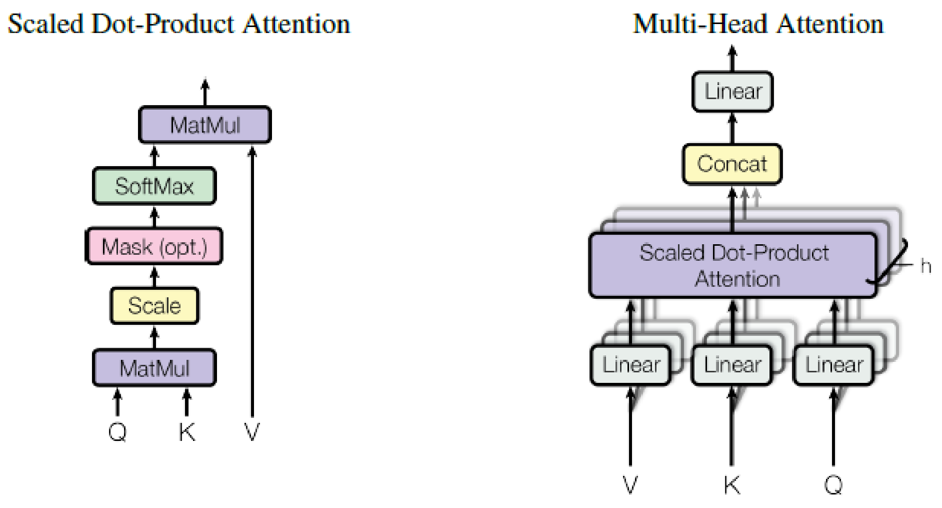


Figure 7.9: Self attention and multiheaded attention [131]

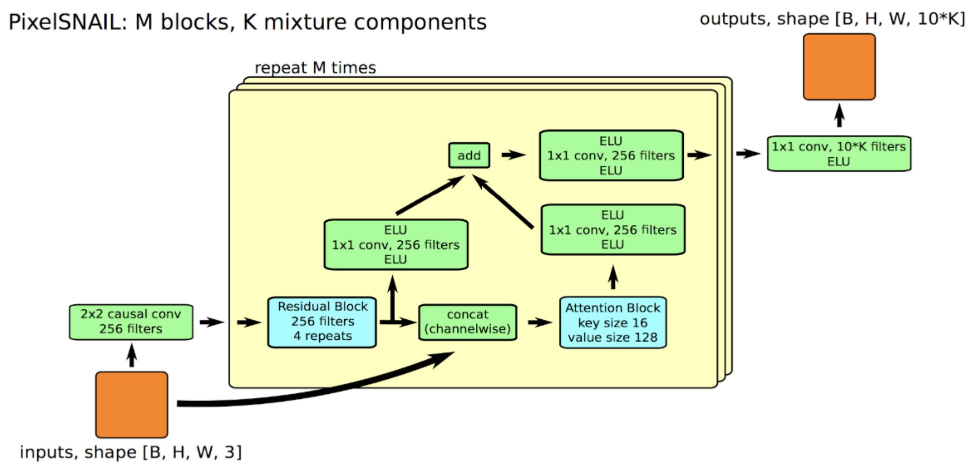

PixelSNAIL: M blocks, K mixture components
outputs, shape [B, H, W, 10*K]
repeat M times
add
ELU 1x1 conv, 256 filters ELU
ELU 1x1 conv, 256 filters ELU
ELU 1x1 conv, 256 filters ELU
1x1 conv, 10*K filters ELU
2x2 causal conv 256 filters
Residual Block 256 filters 4 repeats
concat (channelwise)
Attention Block key size 16 value size 128
inputs, shape [B, H, W, 3]

Figure 7.10: Architecture of Pixel Snail with causal convolutions and attention [24]

This computation can be ensembled to obtain multiple attentions, this is known as multi-headed attention. Figure 7.9 represents the elements of this mechanism.

One example of autoregressive architecture combining causal convolutions and attention is PixelSnail [24]. The goal is to increase the field that a pixel can use to model the autoregressive distribution improving over PixelCNN and PixelCNN++. The main elements of the architecture are a residual block with 2D-causal convolutions that uses masking to enforce autoregressiveness plus the gating mechanism from Gated PixelCNN. Over this an attention block computes the similarity of the sequence at each layer. This architecture takes also other elements from PixelCNN++, like the discretization of the output using a mixture of logistics and a linear autoregressive parameterization to predict the RGB values of a pixel.

A representation of the main elements of the architecture are in figure 7.10.

Another model that uses the transformer architecture is ImageGPT [21]. This is based on the transformer architecture of GPT2 (attention blocks) that only has a decoder module instead of the full transformer. For processing images, these are transformed to low resolution computing discrete tokens with patches of the image that are used for each position and then the image

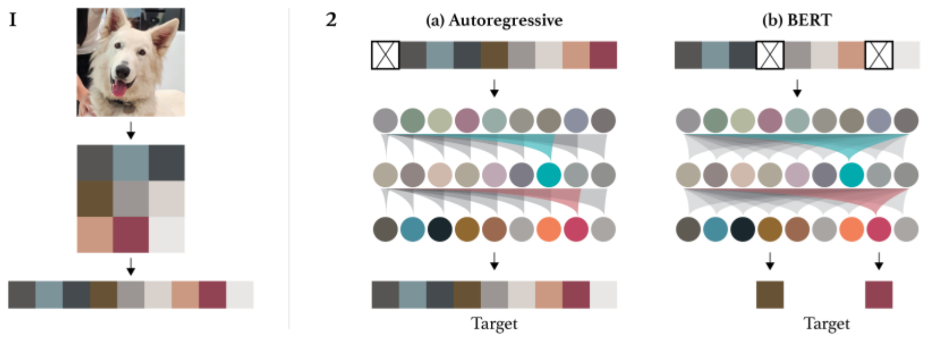


Figure 7.11: Architecture of Pixel Snail with causal convolutions and attention [24]

is linearized. This makes the input sequence as in a language model. As in these models the training can be done autoregressively predicting the last token as the original GPT, or it can also be trained to predict masked tokens as is done in by the BERT mode, predicting arbitrary positions of the sequence (see figure 7.11).

Given the quadratic complexity of attention this model has limitations in the size of the image that can be trained (32×32, 64×64). In the original version they grouped patches of pixels using clustering to increase the size of the vocabulary (512 tokens). This is basically a vector quantization technique where the original tokens are substituted by the cluster centroids. This reduces the sequence lengths proportionally to the size of the patch that is used.

Given that it is trained like a language model (GPT/BERT), this means that the embedding learned by this model can be used as a representation for other tasks or additional tokens can be added for other tasks, like classification.

7.4 Notebook: Autoregressive Models

This notebook shows different autoregressive models

As you already know this kind of models estimate a probability distribution rewritten using the product rule:

$p(x\_1,\ldots,x\_n) = \prod_{i=1}^{n} p(x\_i|x_{<i})$

The network architecture reproduces that autoregressive structire so to obtain samples/probabilites we need to compute iteratively the distribution for each variable.

This notebook uses the `pytorch_generative` library

7.4.1 Auxiliary functions

This funcion performs the training loop and returns the value of the loss function for all the training epochs.

Only a training set is used, you can adapt the code using a test set to monitor the overfitting or terminate the training when it does not improve.

[4]:
```
# Training loop for the different models
def train_loop(model, optimizer, scheduler, loss_fn, dataloader, epochs):
    hist_loss = []
    pbar = tqdm(range(epochs))
    for epoch in pbar:  # loop for every epoch
```

```
        running_loss = 0.0
        for i, data in enumerate(dataloader):
            # get the data and move it to the GPU
            inputs = data.to(device)

            # reset the gradients
            optimizer.zero_grad()

            # apply the model and get the outputs
            outputs = model(inputs)

            # compute the loss
            loss = loss_fn(inputs, outputs)

            # backpropagate the gradients
            loss.backward()
            optimizer.step()

            # store the loss
            running_loss += loss.item()

        if scheduler is not None:
            scheduler.step()
            lr = f'lr: {scheduler.get_last_lr()[0]:.4E}'
        else:
            lr = ''

        hist_loss.append(running_loss / i)
        pbar.set_description(f'loss: {running_loss / i:3.4f}:{lr}')

    return hist_loss
```

7.4.2 MADE (Masked Autoencoder for Density Estimation)

The first model is MADE. This is an autoencoder tha uses masking to define a network representing a probability distribution autoregressively.

We are going to use the digits dataset from `scikit-learn` selecting one of the classes so the training is quick and binarzing the values so we end with a binomial/Bernoulli for each pixel.

[6]:
```
from sklearn.datasets import load_digits
from sklearn.preprocessing import Binarizer
import matplotlib.pyplot as plt
from torch.utils.data import Dataset
```

We select the digit 0 (it can be change to any other digot)

[7]:
```
Xo,y =  load_digits(n_class=10, return_X_y=True)
X = Xo[y==0]
Xb = Binarizer(threshold=7).fit_transform(X)
```

We can visualize the digits befor and after binarization. Changing the threshold we will change the distribution. For this example we have use the value 7.

[8]:
```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.matshow(X[0].reshape(8,8),cmap='gray')
ax1.axis('off')
ax2.matshow(Xb[0].reshape(8,8))
ax2.axis('off')
0;
```

Now we define a dataloader for the training loop. We transform the data to pytorch tensos and we reshape them so they are square matrix (8x8) instead of the original array shape.

[9]:
```
# dataset for the binarized digits
class BinDigits(Dataset):
    def __init__(self, data):
        self.data = torch.reshape(torch.Tensor(data), (-1,8,8)).unsqueeze(1)

    def __getitem__(self, index):
        x = self.data[index]
        return x

    def __len__(self):
        return len(self.data)

digdata = BinDigits(Xb)
bindig_dl = torch.utils.data.DataLoader(digdata, batch_size=32)
```

The loss funtion will be the binary cross entropy. We are learning a binomial distribution for the variables and we want it to be close to the binary values of the pixels.

[10]:
```
def loss_fn(x, preds):
    batch_size = x.shape[0]
    x, preds = x.view((batch_size, -1)), preds.view((batch_size, -1))
    loss = F.binary_cross_entropy_with_logits(preds, x, reduction="none")
    return loss.sum(dim=1).mean()
```

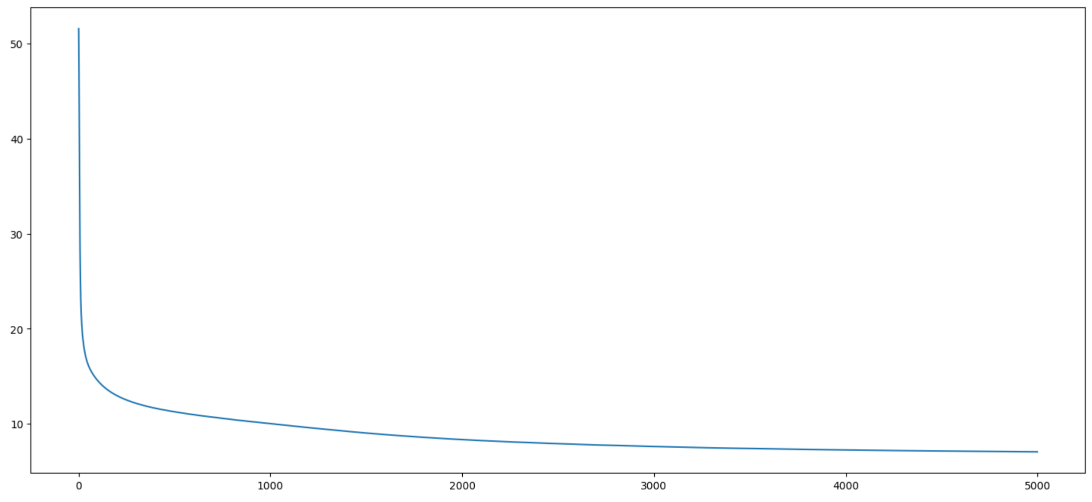
We define a MADE model with input the size of the image and a hidden layer with a number of neurons.

If we increase the number of layers/neurons we will increase the capacity of the model. We should perform a hyperparameter search to find the best fit. We will try with 200 neurons. We will use Adam as optimizer and adjust the learning rate with a multiplicative schedule.

[11]:
```
made_model = models.MADE(input_dim=64, hidden_dims=[200], n_masks=1)
made_model.to(device)
optimizer = optim.Adam(made_model.parameters(), lr=1e-3)
scheduler = lr_scheduler.MultiplicativeLR(optimizer, lr_lambda=lambda _: 0.
  ↪99999)
```

[12]:
```
made_loss = train_loop(made_model, optimizer, scheduler, loss_fn, bindig_dl,␣
  ↪epochs1)
```

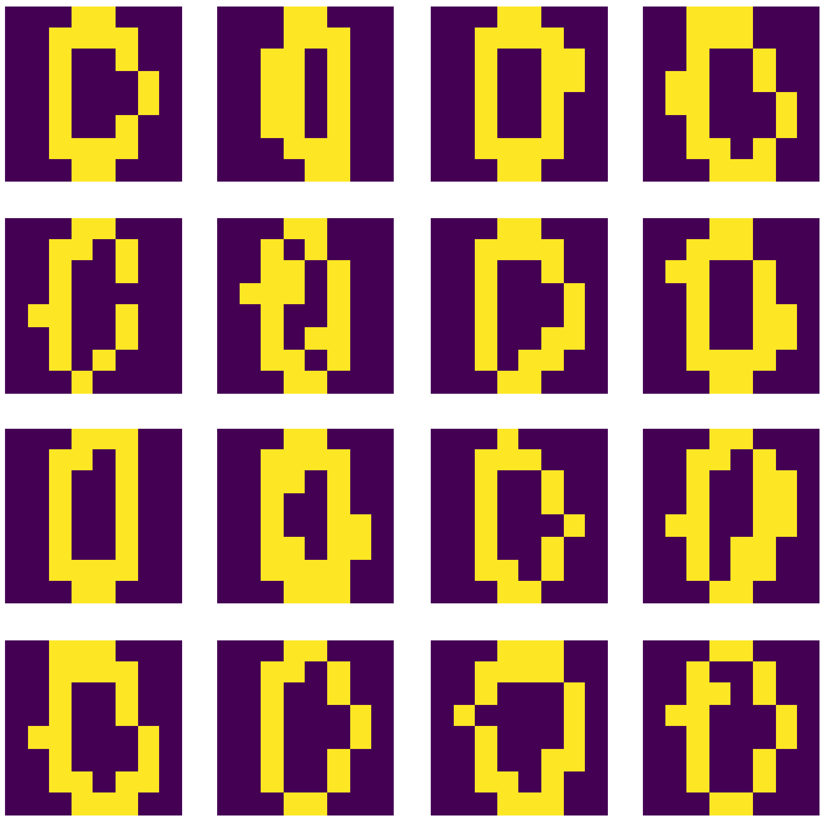
[13]:
```
plt.figure(figsize=(18,8))
plt.plot(made_loss);
```



We can see that the loss function is reduced until almost convergence (obviously we need a test sample to check overfitting).

We can check visually samples from the model to see if they make sense.

[14]:
```
nim = 4
made_samp=made_model.sample(n_samples=nim*nim)
fig = plt.figure(figsize=(6,6))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(made_samp[i].cpu().permute(1,2,0))
    plt.axis('off')
```

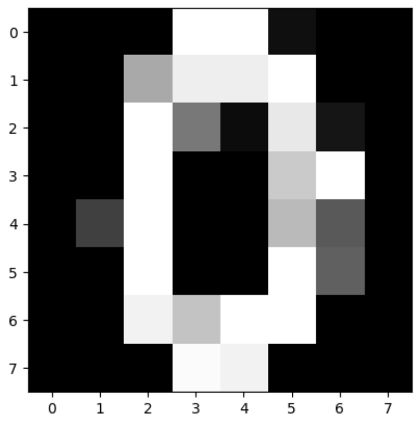


Samples are generated autoregressively pixel by pixel computing their probability and sampling accordingly to its value.

We can generate an unlimited number of samples just running the model.

Given that we have a probabilistic model we can compute the samples probability. The model computes the logits for each pixel. Applying the sigmoid function we can obtain the probabilty for each pixel (probability of being 1)

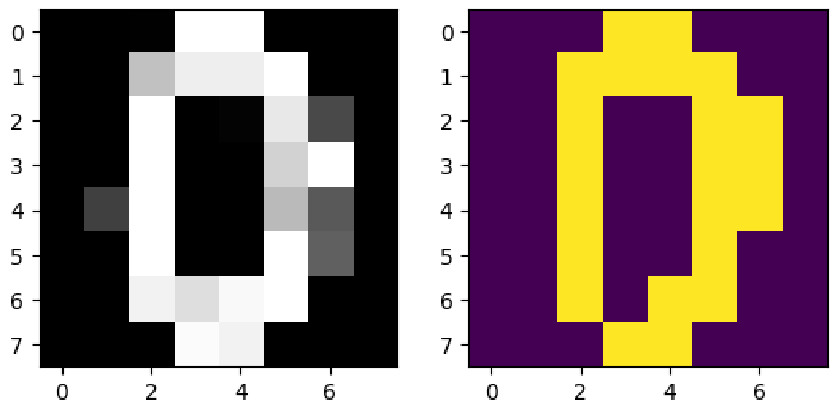
```
[15]: lg = torch.sigmoid(made_model(made_samp[2]).detach())
      plt.imshow(lg.cpu().permute(1,2,0), cmap='gray');
```



We can also evaluate training samples

```
[16]: lg = torch.sigmoid(made_model(torch.tensor(Xb[0].reshape((-1,8,8)),␣
       ↪dtype=torch.float32).to(device)).detach())
```

```
plt.subplot(1, 2, 1)
plt.imshow(lg.cpu().permute(1,2,0), cmap='gray');
plt.subplot(1, 2, 2)
plt.imshow(Xb[0].reshape(8,8));
```



```
[17]: # Negative log likelihood for the bernoulli distribution
def nll(model, samp):
    epsilon = 1e-15
    snll = 0.0
    for i in range(samp.shape[0]):
        mu = torch.sigmoid(model(samp[i]).detach())
        snll += ((samp[i] * torch.log(mu+epsilon)) + ((1-samp[i]) * torch.
 ↪log(1-mu+epsilon))).sum()
    return -snll.cpu()
```

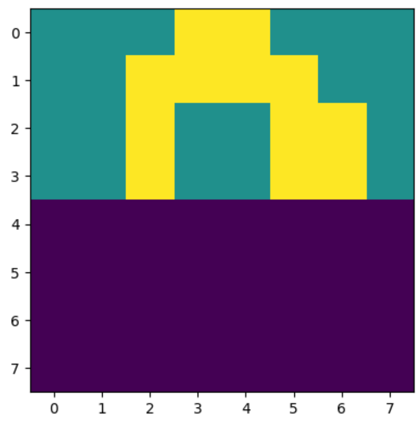
This is the log likelihood for the first 100 samples

```
[18]: nll(made_model, torch.tensor(Xb[:100].reshape((-1,8,8)), dtype=torch.
 ↪float32).unsqueeze(1).to(device))
```

```
[18]: tensor(601.3405)
```

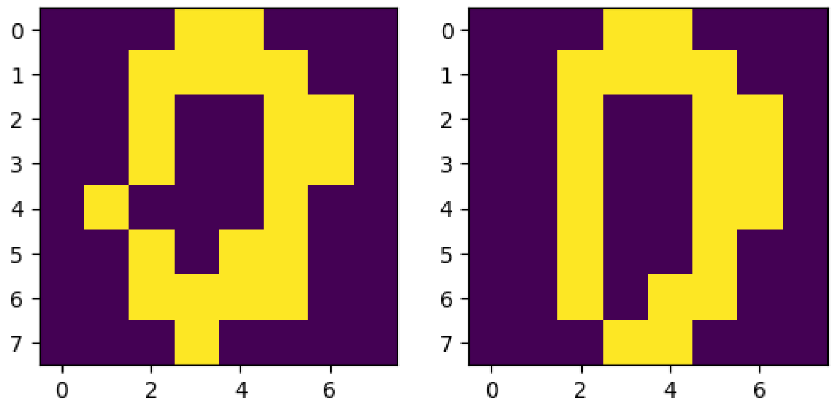
Other advantage of an autoregressive model is that we can complete partial samples. For instance, we can use a training sample, mask a part of the values and obtain the missing part conditining the model with the visible part.

```
[19]: cond = Xb[0].copy()
cond[32:]=-1
plt.imshow(cond.reshape((8,8)));
```



Obviously, sampling many time we will obtain different completions. We can comapre to the original sample.

[20]:
```
csamp=made_model.sample(n_samples=1,conditioned_on=torch.reshape(torch.
 ↪Tensor(cond), (-1,8,8)).unsqueeze(1).to(device))
plt.subplot(1, 2, 1)
plt.imshow(csamp[0].cpu().permute(1,2,0));
plt.subplot(1, 2, 2)
plt.imshow(Xb[0].reshape(8,8));
```



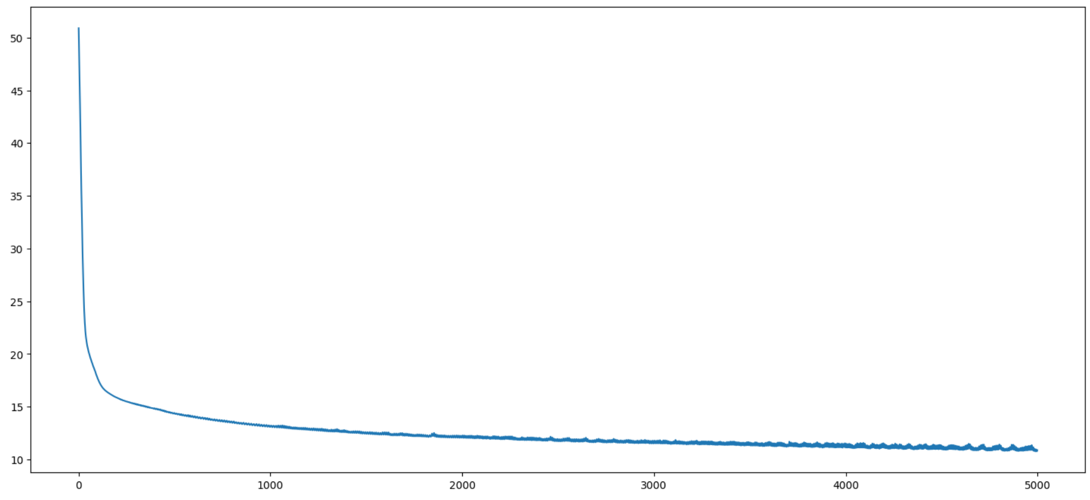
7.4.3 Pixel CNN

Now we will fit a PixelCNN to the same data. We have chosen a set of hyperparameters, but We should perform a hyperparameter search to obtain optimal results.

[21]:
```
pcnn_model = models.PixelCNN(in_channels=1, out_channels=1, n_residual=2,␣
 ↪residual_channels=6, head_channels=12)
pcnn_model.to(device)
optimizer = optim.Adam(pcnn_model.parameters(), lr=1e-3)
scheduler = lr_scheduler.MultiplicativeLR(optimizer, lr_lambda=lambda _: 0.
 ↪99999)
```

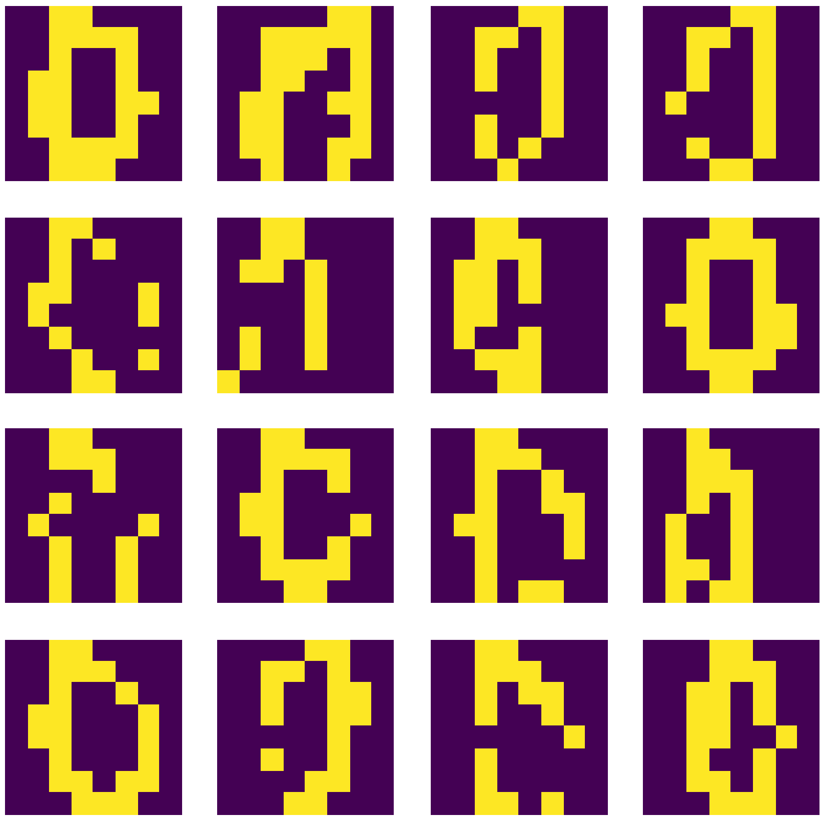
[22]:
```
pcnn_loss = train_loop(pcnn_model, optimizer, scheduler, loss_fn, bindig_dl,␣
 ↪epochs1)
```

[23]:
```
plt.figure(figsize=(18,8))
plt.plot(pcnn_loss);
```



Now we can sample from the model

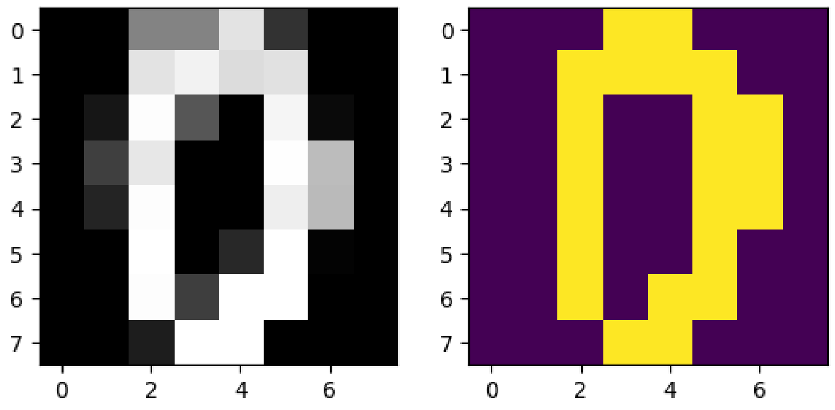
[24]:
```
nim = 4
pcnn_samp=pcnn_model.sample(n_samples=nim*nim)
fig = plt.figure(figsize=(6,6))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(pcnn_samp[i].cpu().permute(1,2,0))
    plt.axis('off')
```



We can compare the probabilities assigned now to the first sample to the ones computed by

MADE.

[25]:
```
lg = torch.sigmoid(pcnn_model(torch.tensor(Xb[0].reshape((-1,8,8)),␣
  ↪dtype=torch.float32).to(device)).detach())
plt.subplot(1, 2, 1)
plt.imshow(lg.cpu().permute(1,2,0), cmap='gray');
plt.subplot(1, 2, 2)
plt.imshow(Xb[0].reshape(8,8));
```



and the NLL of the first 100 samples (adjusting the hyperparameters better probably would reduce its value)

[26]:
```
nll(pcnn_model, torch.tensor(Xb[:100].reshape((-1,8,8)), dtype=torch.
  ↪float32).unsqueeze(1).to(device))
```

[26]:
```
tensor(959.4286)
```

7.4.4 Gated Pixel CNN

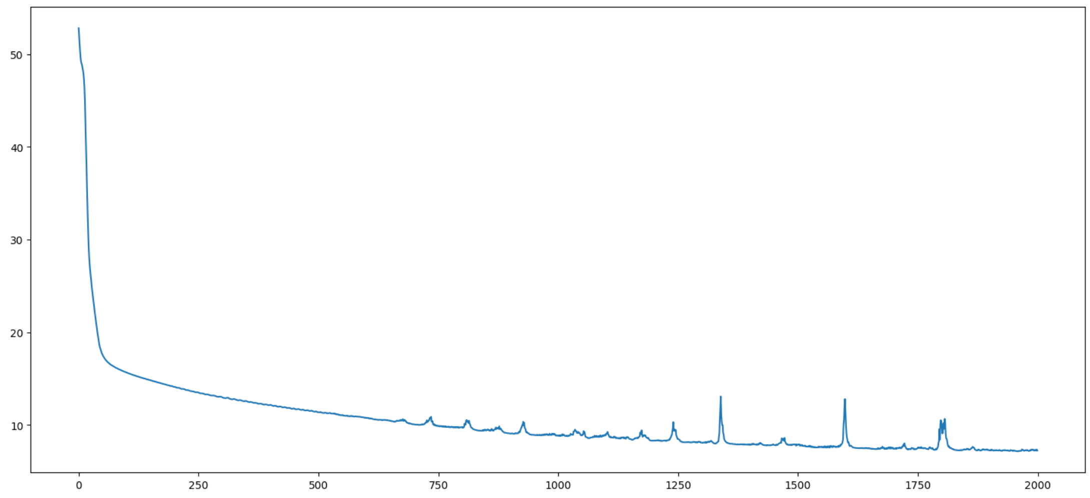
PixelCNN uses 1D and 2D convolutions to avoid losing information each layer, we can check is performance with this data

Now we have additional hyperparameters, gated blocks (n_gated) with certain number of channels (gated_channels) and the channels for the 1x1 convoltions that genrate the output. We have decided for a set of hpyper parameters, but a hyperparameter search should be performed.

[27]:
```
gpcnn_model = models.GatedPixelCNN(in_channels=1, out_channels=1, n_gated=4,␣
  ↪gated_channels=8, head_channels=8)
gpcnn_model.to(device)
optimizer = optim.Adam(gpcnn_model.parameters(), lr=1e-3)
scheduler = lr_scheduler.MultiplicativeLR(optimizer, lr_lambda=lambda _: 0.
  ↪99999)
```

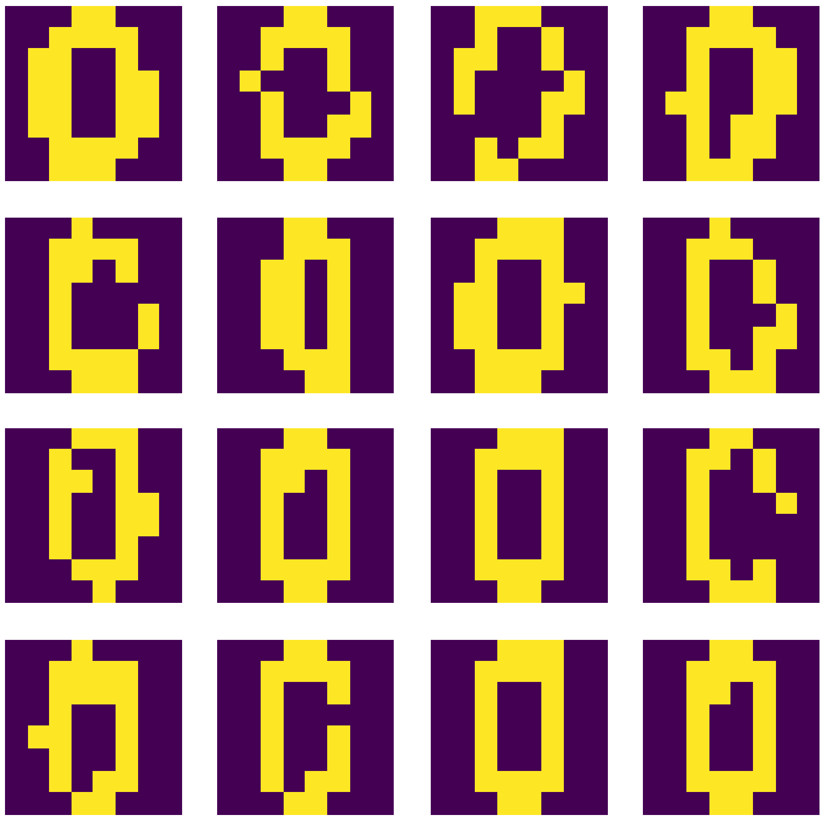
[28]:
```
gpcnn_loss = train_loop(gpcnn_model, optimizer, scheduler, loss_fn,␣
  ↪bindig_dl, epochs2)
```

[29]:
```
plt.figure(figsize=(18,8))
plt.plot(gpcnn_loss);
```

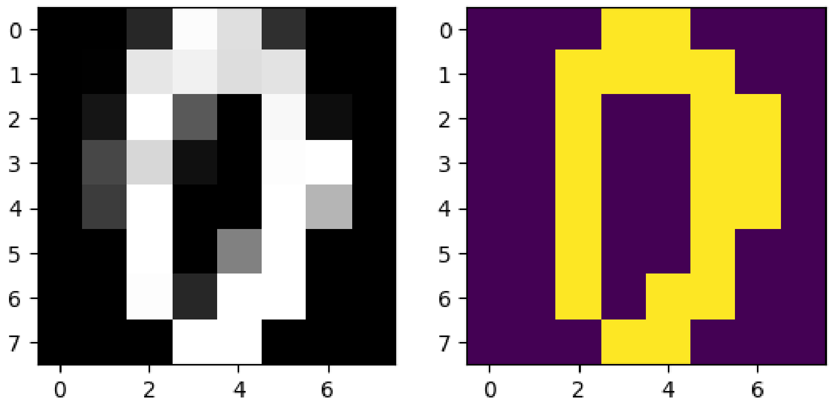


Quality of the samples will depend on the capacity of the network and the training time.

```
[30]: nim = 4
gpcnn_samp=gpcnn_model.sample(n_samples=nim*nim)
fig = plt.figure(figsize=(6,6))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(gpcnn_samp[i].cpu().permute(1,2,0))
    plt.axis('off')
```



[31]:
```
lg = torch.sigmoid(gpcnn_model(torch.tensor(Xb[0].reshape((-1,8,8)),␣
 ↪dtype=torch.float32).unsqueeze(1).to(device)).detach())

plt.subplot(1, 2, 1)
plt.imshow(lg.squeeze(1).cpu().permute(1,2,0),cmap='gray');
plt.subplot(1, 2, 2)
plt.imshow(Xb[0].reshape(8,8));
```



[32]:
```
def nll2(model, samp):
    epsilon =1e-15
    snll = 0.0
    for i in range(samp.shape[0]):
        mu = torch.sigmoid(model(samp[i].unsqueeze(1)).detach())
        snll += ((samp[i] * torch.log(mu+epsilon)) + ((1-samp[i]) * torch.
 ↪log(1-mu+epsilon))).sum()
    return -snll.cpu()

nll2(gpcnn_model, torch.tensor(Xb[:100].reshape((-1,8,8)), dtype=torch.
 ↪float32).unsqueeze(1).to(device))
```

[32]:
```
tensor(605.3003)
```

NLL is not the best, with more gated blocks and more training time results should improve.

7.4.5 Image GPT

Image GPT uses a transformer architecture, it works with an embedding of the image predicting tokens, but in this implementation the prediction is done at pixel level working with the image as a sequence.

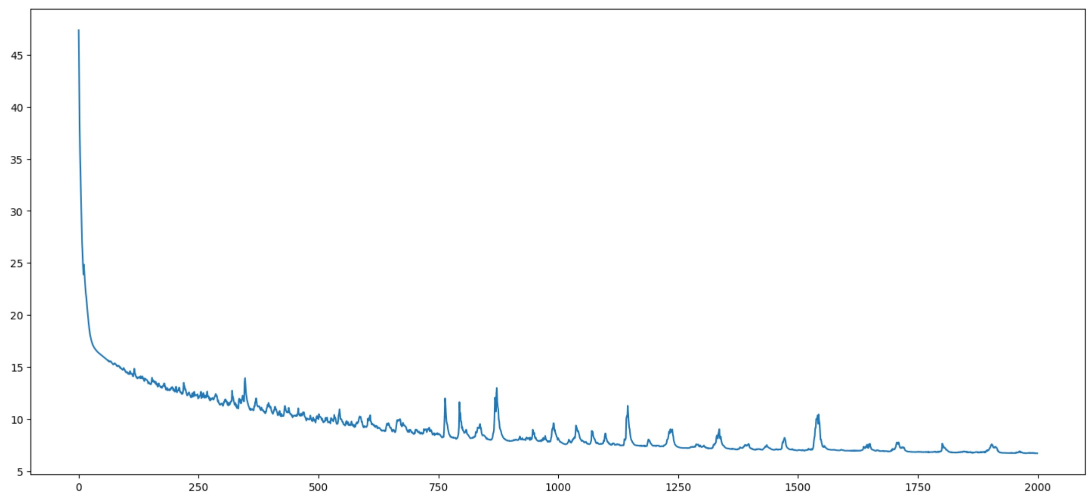
The architecture defines then umber of trnasformer blocks, the number of heads and the number of channels in the embedding of the transformer.

Hyperparameters have been chosen manually, a more exhaustive exploration should improve results.

```
[33]: igpt_model = models.ImageGPT(in_channels=1, out_channels=1, in_size=8,
              n_transformer_blocks=3,
              n_attention_heads=8,
              n_embedding_channels=8)
      igpt_model.to(device)
      optimizer = optim.Adam(igpt_model.parameters(), lr=5e-3)
      scheduler = lr_scheduler.MultiplicativeLR(optimizer, lr_lambda=lambda _: 0.
       ↪999)
```

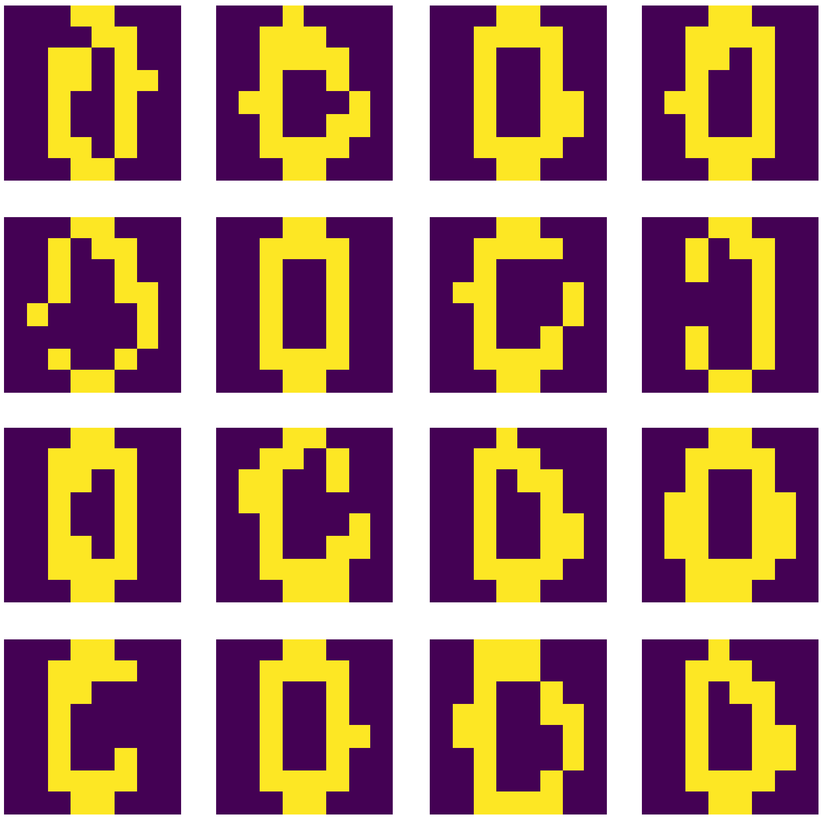
```
[34]: igpt_loss = train_loop(igpt_model, optimizer, scheduler, loss_fn, bindig_dl,␣
       ↪epochs2)
```

```
[35]: plt.figure(figsize=(18,8))
      plt.plot(igpt_loss);
```

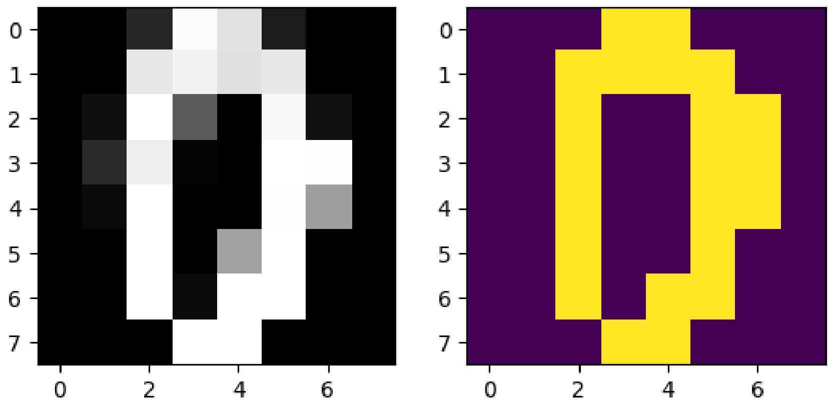


Samples maybe (?) seem better

```
[36]: nim = 4
      pcnn_samp=igpt_model.sample(n_samples=nim*nim)
      fig = plt.figure(figsize=(6,6))
      for i in range(nim*nim):
          plt.subplot(nim, nim, i+1)
          plt.imshow(pcnn_samp[i].cpu().permute(1,2,0))
          plt.axis('off')
```



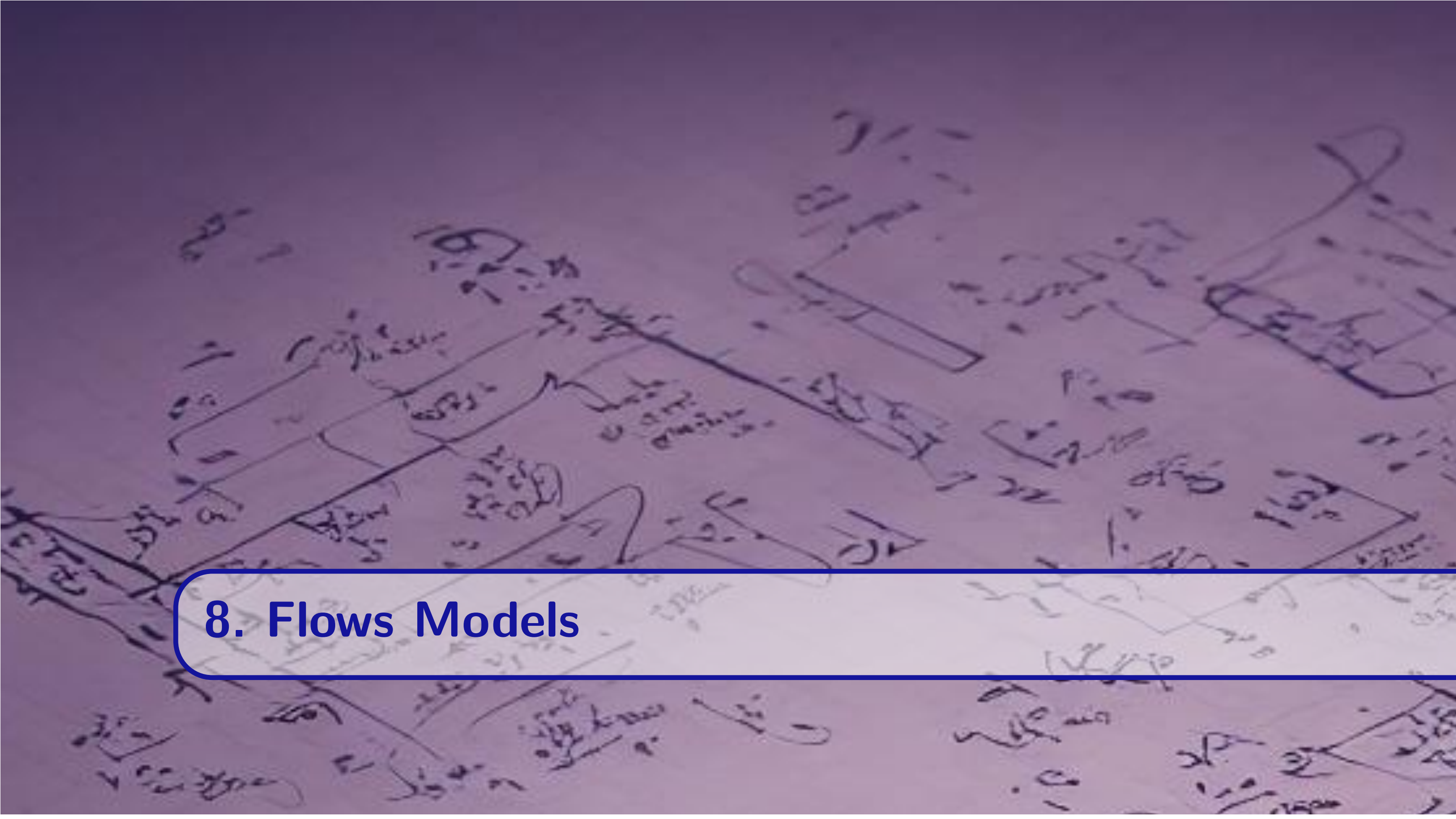
```
[37]: lg = torch.sigmoid(igpt_model(torch.tensor(Xb[0].reshape((-1,8,8)),␣
       ↪dtype=torch.float32).unsqueeze(1).to(device)).detach())

      plt.subplot(1, 2, 1)
      plt.imshow(lg.squeeze(1).cpu().permute(1,2,0),cmap='gray');
      plt.subplot(1, 2, 2)
      plt.imshow(Xb[0].reshape(8,8));
```



NLL has improved.

```
[38]: nll2(igpt_model, torch.tensor(Xb[:100].reshape((-1,8,8)), dtype=torch.
       ↪float32).unsqueeze(1).to(device))
```

```
[38]: tensor(555.4658)
```



8. Flows Models

8.1 Modeling probability distributions

One caveat of the autoregressive models that were explained on the previous chapter is that they model discrete probability distributions. We should be able to model also continuous distributions in order to approach more general problems. The idea is also to model the distribution of data with a function that fulfills the properties of probability distributions. Also, we want to use maximum likelihood for fitting its parameters.

We have already seen a model that is able to fit a continuous probability distribution to data, the Gaussian Mixture model. Unfortunately, this model does not work well for natural and high dimensional data. Sampling from this distribution involves picking a centroid from the mixture according to its weight and then adding gaussian noise distributed as the fitted covariance matrix. This is problematic for instance on the images' domain where this sampling usually produces blurry images.

One alternative to framework for fitting arbitrary distributions to data are **flows**. This is a general method for fitting distributions based on changing a density model into another through a chain of transformation functions $f_\theta = f_1, \ldots, f_n$. That chain of transformations maintains the properties of a density function each step. Basically they adapt the density of a chosen distribution to model another distribution.

Applying this chain of functions to a value x from a dataset X will give another data z ($z = f_\theta(x)$) that will belong to a distribution $p_Z(z)$ that we can choose arbitrarily. By the behaviour of the transformation, fixing the distribution Z will make X also to be distribution. In the case of using the normal distribution $\mathcal{N}(0,1)$ as the target distribution, we have what are called *normalizing flows*.

For instance these kinds of transformations include the quantile and power transformations that we explained on the chapter of data preprocessing. In these cases, we only use one function in the chain and the functions are predefined to have a specific form, but their effect is to transform the probability mass of the data to fit the probability mass of the normal distribution.

This chain of functions can not be arbitrary functions. In order to maintain the properties of probability distributions they must fulfill three constraints:

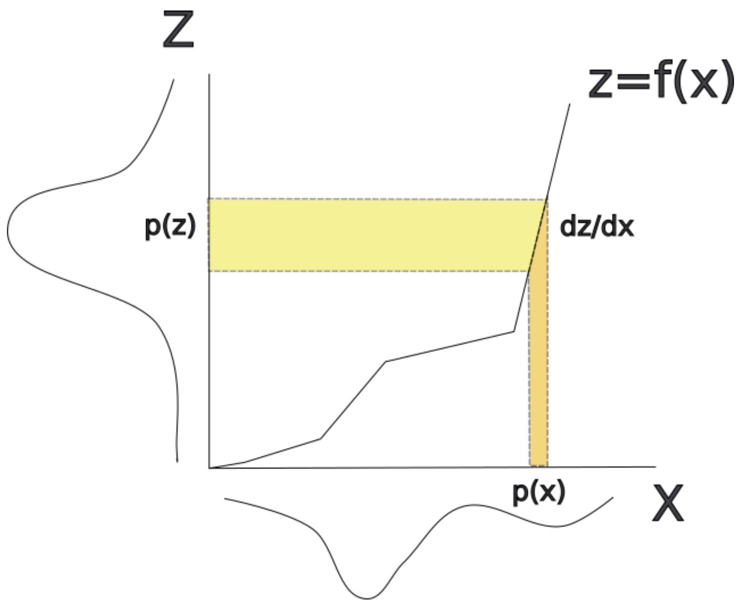


Figure 8.1: Change of variable using a flow that transforms between probability masses of distribution X and Z

- They must be **monotone functions** so, they map uniquely the elements from a function to another, they are bijective
- They must be **differentiable functions** so, it is possible to train the parameters of the functions with gradient descent using maximum likelihood
- They must be **invertible functions** in order to be able to obtain samples from $p(z)$ and transform them to samples of $p(x)$, using $x = f_\theta^{-1}(z)$

8.2 Learning flows

When learning a flow we will not have direct access to the probability distribution of the data ($p_\theta(x)$) for the task of optimizing its log likelihood since it is unknown, but we can access to samples from the target distribution of the flow (for instance, the gaussian distribution). Since the function that defines the flow f_θ is a monotonous function we can apply the change of variable formula $z = f_\theta(x)$ that will allow us to write the derivatives of the probability distribution from the perspective of the Z distribution:

$$p_\theta(x)dx = p_Z(z)dz = p_Z(f_\theta(x))dx = p_Z(f_\theta(x))\left|\frac{\partial f_\theta(x)}{\partial x}\right|$$

Basically, what this equivalence is means is that the derivative on one side is scaled by the derivative of the transformation function (the flow). What it is happening is that the probability mass of one distribution is being scaled down or up to adapt to the probability mass of the other distribution maintaining a bijective mapping. This bijection allows to map samples from one distribution to the other. This is represented on figure 8.1

Since we can generate samples from one distribution given the other, we can rewrite the likelihood of $p_\theta(x)$ as a function of samples from distribution Z, transforming the log likelihood as:

$$\max_\theta \sum_i \log p_\theta(x_i) = \max_\theta \sum_i \log p_z(f_\theta(x_i)) + \log\left|\frac{\partial f_\theta(x_i)}{\partial x}\right|$$

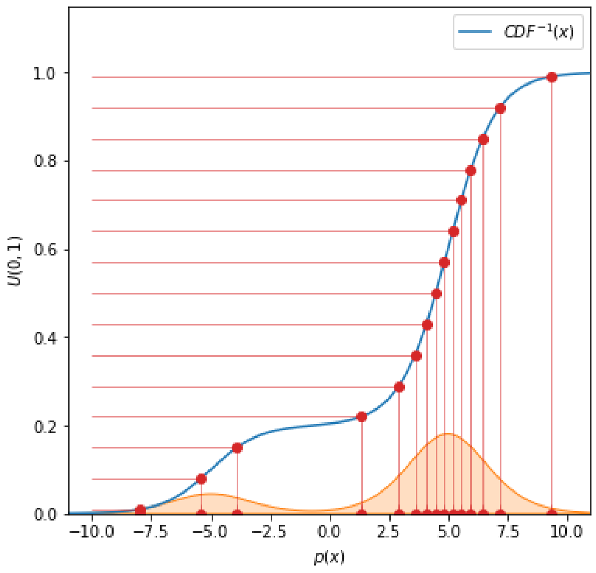


Figure 8.2: The CDF of a distribution maps to the uniform distribution and holds the properties of a flow

Flows are a general framework for transforming probability distributions and allow performing the transformation between any pair of smooth probability distribution functions. To show how this is possible, consider the cumulative distribution function of any distribution (see figure 8.2), this is a monotone, differentiable and invertible function that maps any distribution to the uniform distribution. The composition of flows is also a flow, this means that composing the cumulative distribution function of two arbitrary distributions having as intermediate the uniform distribution we can transform samples from one into the other.

In order to learn an arbitrary flow we will use a neural network. Given the constraints of flows not all networks are possible. We will be constrained to networks with activation functions that are invertible, for instance the sigmoid, the tanh, the LeakyReLU, but not the ReLU since it is not bijective. Also, the dimensionality of each layer must be the same since we must maintain the total probability mass of the distributions. Since the composition of flows is also a flow we can use several layers that will learn each one an individual flow.

8.2.1 Autoregressive Flows

For computing the joint probability distribution when there are more than one dimension on our probability distribution we can resort to the same strategy as on the previous chapter. We can define an autoregressive distribution defining a flow transformation as:

$$z_i = f_\theta(x_i|x_{<i}) \quad x_i = f_\theta^{-1}(z_i|x_{<i})$$

The log likelihood is defined as before, but each variable adds a term $p_x(x) = \prod_{i=1}^{D} p_x(x_i|x_{<i})$ to the likelihood, obtaining:

$$\max_\theta \sum_i \sum_j \log p_z(f_\theta(x_{ij}|x_{<j})) + \log \left| \frac{\partial f_\theta(x_{ij}|x_{<j})}{\partial x_j} \right|$$

In this case the training of the flow can be done in parallel since we have all the values of x, but as in the previous autoregressive models, inference needs to be done variable by variable, computing all the values $z_{<j}$ before obtaining z_i. This makes this model impractical for domains of high dimensionality.

An alternative are the Inverse Autoregressive Flows (IAF), since flows are invertible functions, we can reformulate the inverse problem as:

$$z_i = f_\theta^{-1}(x_i|z_{<i}) \quad x_i = f_\theta(z_i|z_{<i})$$

This moves the burden of the autoregressive computation to the training and allows fast inference. This is what is used by some models as Parallel Wavenet [105], an enhancement over the Wavenet autoregressive model that appeared on the previous chapter. In this case the training is performed by model distillation from a trained Wavenet.

8.2.2 Multidimensional Flows

Autoregressive flows only allow changing one variable at a time, but there are no constraints about what dimensions a flow can have. When operating with more than one dimension instead of scaling up or down the values of one variable we will adapt multidimensional volumes distributing the probability mass from distribution x to fill the probability mass of distribution z, this rewrites the relationship between the distributions as:

$$p(x)vol(dx) = p(z)vol(dz) \Rightarrow p(x) = p(z)\frac{vol(dz)}{vol(dx)} = p(z)\left|det\frac{dz}{dx}\right|$$

This also rewrites the change of variable formula as:

$$p_\theta(x) = p(f_\theta(x))\left|det\frac{\partial f_\theta(x)}{dx}\right|$$

In this case the maximum likelihood optimization is defined as:

$$\min_\theta \mathbb{E}_x[-\log(p_\theta(x)] = \mathbb{E}_x\left[-\log p_z(f_\theta(x)) - \log\left|det\frac{\partial f_\theta(x)}{dx}\right|\right]$$

This formulation is scalable as long as the determinant of the jacobian is easy to compute, meaning that it can not be a full matrix. Only flows that have diagonal or triangular jacobians are then scalable, since the determinant correspond to the product of the elements of the diagonal. This reduces the expressive power of a multidimensional flow. One way of alleviating this problem is by the composition of several flows.

There are several kinds of multidimensional flows, the simplest one corresponds to the *affine flow* that is defined by an affine transformation (basically a linear transformation, as in a one layer perceptron). This corresponds to the transformation:

$$f(x) = A^{-1}(x-b)$$

Sampling can be done using the Gaussian distribution (normalizing flow) obtaining:

$$x = Az + b$$

Computing the log likelihood of this model will be expensive for high dimensionality if the jacobian matrix is not diagonal or triangular. A simplification of this model is the *element wise flow*, basically a transformation is performed for each variable independently $f_\theta(x_1,\ldots,x_n) = (f_\theta(x_1),\ldots,f_\theta(x_n))$ this makes the jacobian diagonal with a determinant computed as:

$$\begin{aligned}\frac{\partial z}{\partial x} &= diag(f'_\theta(x_1),\ldots,f'_\theta(x_n))\\ det\frac{\partial z}{\partial x} &= \prod_{i=1}^{d} f'_\theta(x_i)\end{aligned}$$

This is more efficient, but the expressive power of this model is drastically reduced.

We can combine these two ideas on a more expressive model by splitting the variables of the distribution in two groups $x_1, \ldots, x_{\frac{d}{2}}, x_{\frac{d}{2}+1}, \ldots, x_d$. One half will be maintained as is, and the other half will be passed through an affine transformation obtaining two sets of variables:

$$\begin{aligned} z_{1:\frac{d}{2}} &= x_{1:\frac{d}{2}} \\ z_{\frac{d}{2}+1:d} &= x_{\frac{d}{2}+1:d} \cdot s_\theta(x_{1:\frac{d}{2}}) + t_\theta(x_{1:\frac{d}{2}}) \end{aligned}$$

This is called a **coupling layer** and constitutes an invertible transformation, since we are only translating and scaling the variables. The functions t_θ and s_θ can be arbitrary neural networks. This makes the Jacobian:

$$\frac{\partial z}{\partial d} = \begin{bmatrix} I & 0 \\ \frac{\partial z_{\frac{d}{2}+1:d}}{\partial x_{1:\frac{d}{2}}} & diag(s_\theta(x_{1:\frac{d}{2}})) \end{bmatrix}$$

Given that is a triangular matrix the determinant can be computed as:

$$det\frac{\partial z}{\partial d} = \prod_{i=1}^{d} s_\theta(x_{1:\frac{d}{2}})_i$$

A coupling layer can be composed in a deep network where the subset of variables for each partition can be changed on each layer using a permutation later. A permutation is an invertible operation that does not contribute to the jacobian since it is fixed for the training. This permutation allows intertwining the variables, increasing the expressive power of the flow by mixing their contribution to the other variables. The sequence of coupling layers does not affect to the cost of the computation of the Jacobian of the whole flow, since it is the product of the Jacobians of each layer.

This transformation is used by the models NICE (Non-linear Independent Components Estimation) [33] that only uses translations and RealNVP (Non Volume Preserving) [34] that uses translation and scaling. The non volume preserving refers to the fact that using scaling increases/reduces the volume of the probability distribution.

RealNVP is used for image generation, that allows for a more specific mixing of variables. This model applies a checkerboard pattern for the partition of the variables spatially and channel wise, combining these patterns on the different coupling layers. It also uses a multiscale architecture where on certain layers the spatial dimension is squeezed in half multiplying the number of channels by four. This maintains the total number of variables as is needed in flow based architectures. The final architecture combines blocks composed by two spatial alternate checkerboard patterns, a squeeze operation and two channel wise alternate checkerboard patterns with residual connections among blocks.

8.2.3 Non linear flows

All the affine flows that we have seen are limited in their expressive power and even when many layers are used there are distributions that can not be captured. In the recent literature there are many proposals for overcoming that limitation.

GLOW [80] follows on the steps of RealNVP, but adressing one of its drawbacks, the permutation layer. These permutations are fixed and do not adapt to the needs of the flow. This architecture introduces 1×1 invertible convolution as an alternative to that layer.

The 1 convolution is an invertible transformation when the number of input and output channels is the same and corresponds to a generalization of a permutation matrix. The computational cost of the Jacobian can be reduced using a LU decomposition of the matrix that performs the convolutions. The matrix of weights W is initialized to a rotation matrix and decomposed to:

$$W = PL(U + diag(s))$$

where P is a permutation matrix that is fixed, L is a lower triangular matrix with a diagonal of ones, U is a upper diagonal matrix with zeros in the diagonal and s is a vector. The learnable parameters are L, U and s. Since P is fixed, it does not affect to the jacobian, the jacobian of L is one and the jacobian of $U + diag(s)$ is the norm of s. T

The GLOW architecture combines blocks of activation normalization channels to stabilize the training (makes activation to have mean 0 and variance 1), 1×1 convolutions and affine transformations in the same fashion as RealNVP (splitting the variables). This network was one of the first efficient flows that allowed scaling to images of a resolution up to 256. It also allows for the manipulation of the latent representation using interpolation and finding latent directions that allow to change characteristics of the images. You can see some examples on the web page of the paper (`https://openai.com/blog/glow/`).

Another group of methods is related to different kinds of convolutions that are invertible. They have the advantage of being particular cases of the general convolutional operator, so they can be implemented efficiently and have more expressive power than affine transformations. This includes for example circular and symmetrical convolutions [74] or the periodic convolution [69] that have a Jacobian that can be computed efficiently using the Fourier transform. These flows are defined by matrices with special properties that allow efficient computation or approximation of the jacobian, that had lead to other kinds of special matrices that can be used for defining the flow, like, for instance, the Sylvester flow [70] that uses the Sylvester identity $(det(I + AB) = det(I + BA))$ that allows the computation of the jacobian as a recurrence.

Another way of obtaining non-linear flows is to use piecewise functions that are differentiable up to a specific order. This includes for instance functions defined as different kinds of splines whose coefficients can be computed using a neural network. For instance Neural Spline Flows [40] use rational quadratic polynomials fitted to the density functions according to a set of bins. These functions can be analytically inverted and their parameters and the bins for dividing the density are computed by a neural network.

8.3 Notebook: Flows

This notebook shows different flow models. These models allow transforming a probability distribution into another using a function that is invertible (apart from differentiable and monotone).

This function allows estimating and sample a complex distribution from its correspondence with a more simple distribution.

We are going to use the `normflows` library.

8.3.1 Auxiliary functions

This function performs the training loop and returns the loss for all the training. The loss used is the Kullbak-Leibler divergence.

This function does not use a validation sample to monitor the training or to perform early termination, feel free to improve it.

[4]:
```
# Training loop for the different models
def train_loop(model, optimizer, scheduler, dataloader, epochs,␣
 ↪conditional=False):
    hist_loss = []
    pbar = tqdm(range(epochs))
    for epoch in pbar:
        running_loss = 0.0
        for i, data in enumerate(dataloader):
            optimizer.zero_grad()

            # compute loss depending on whether it is conditional or not
            if not conditional:
                loss = model.forward_kld(data.to(device))
            else:
                loss = model.forward_kld(data[0].to(device), data[1].
 ↪to(device))

            # if the loss is not nan or inf, do backpropagation
            if ~(torch.isnan(loss) | torch.isinf(loss)):
                loss.backward()
                optimizer.step()

            # store loss
            running_loss += loss.item()

        if scheduler is not None:
            scheduler.step()
            lr = f'lr: {scheduler.get_last_lr()[0]:.4E}'
        else:
            lr = ''

        hist_loss.append(running_loss / i)
        pbar.set_description(f'loss: {running_loss / i:3.4f}:{lr}')

    return hist_loss
```

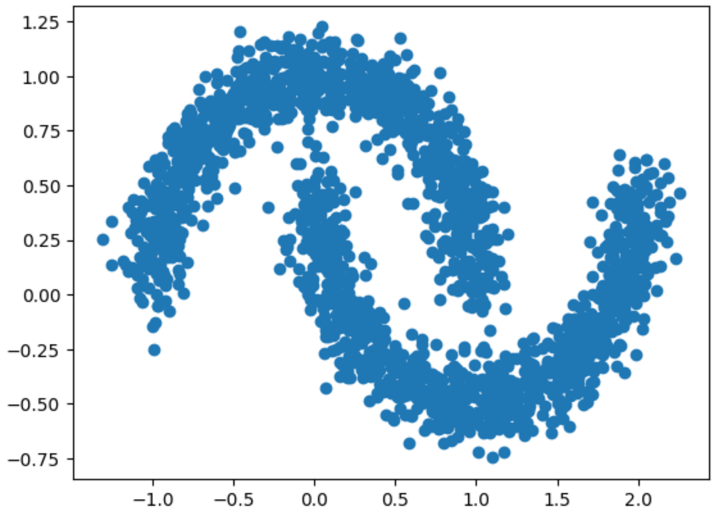
8.3.2 Once in a blue moon

As a toy sample we will transform a dataset that has one distribution into a simpler one (gaussian) and from that model we will genrate samples using the inverse transformation.

For that we will generate samples from the two moons dataset that has to crescent moons intertwined (not linearly separable), so we have a bimodal distribution that we will model using a gaussian distribution.

[6]:
```
X,_ = make_moons(2048, noise=0.1)
```

[7]:
```
plt.scatter(X[:,0], X[:,1]);
```



Now we define a flow that has as base distribution a bidimensional gaussian and use a sequence of transformations composed by affine flows and permutation layers. The affine flows compute translations and scalings using a two layer MLP. This flows follows the transformation used by RealNVP applying the transformation to half of the variables (just one variable) and leaving untouched the other half (the other variable). The permutation layer swaps the transformed variables.

The number of layers is selected so the flow has enough expressiveness.

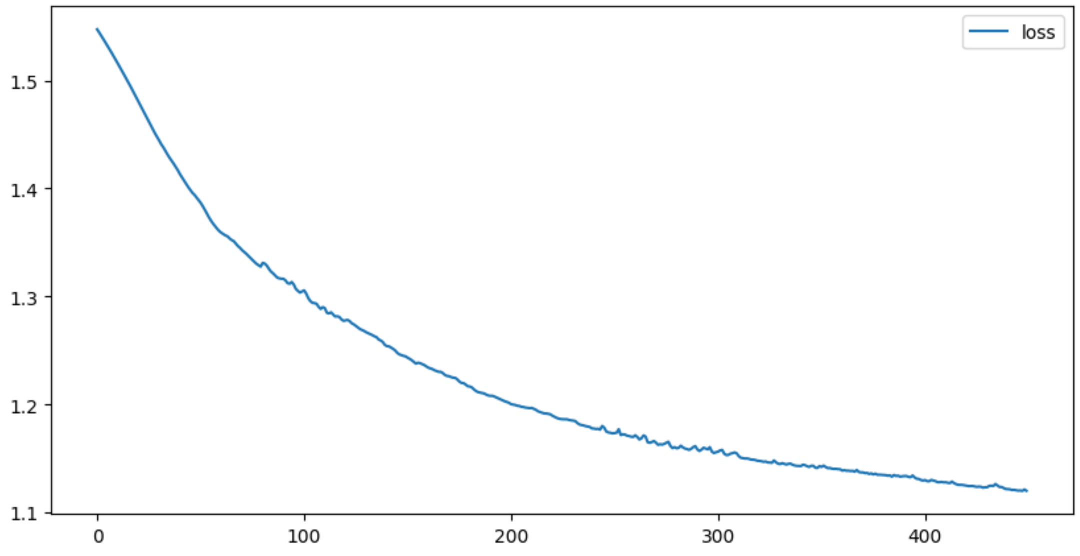
```
[8]: base = nf.distributions.base.DiagGaussian(2)

     # Define a list of layers for the flow
     num_layers = 5
     flows = []
     for i in range(num_layers):
         # The parameters of the Affine Coupling Flow are calculated with a␣
      ↪neural network
         # Each layer transforms half of the parameters and calculates the␣
      ↪translation and scaling
         # The input size of the network is #variables/2 and the output is the␣
      ↪scaling and translation
         param_map = nf.nets.MLP([1, 32, 32, 2], init_zeros=True)
         # Affine Coupling layer
         flows.append(nf.flows.AffineCouplingBlock(param_map))
         # Permutation layer
         flows.append(nf.flows.Permute(2, mode='swap'))

     # We define the flow from the base distribution and the flow layers
     model_2m = nf.NormalizingFlow(base, flows).to(device)
```

This is a dataloader

```
[9]: moons_dl = torch.utils.data.DataLoader(torch.tensor(X).type(torch.float32),␣
      ↪batch_size=128)
```

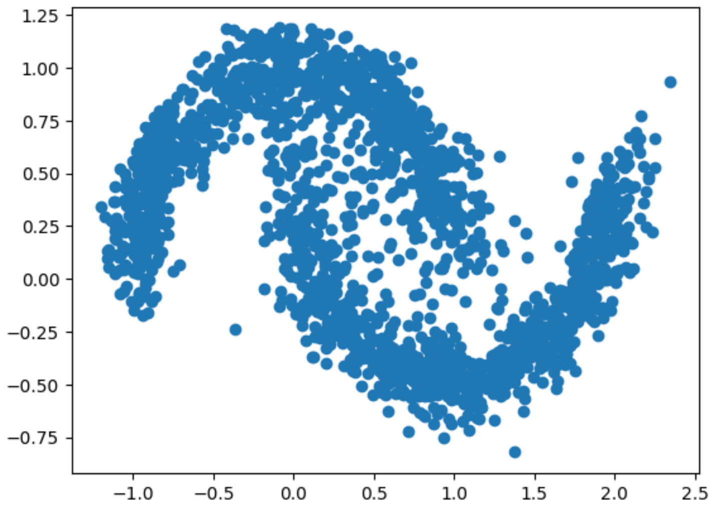
and we optimize the model for enough epochs to converge

```
[10]: optimizer = torch.optim.Adamax(model_2m.parameters(), lr=1e-3,␣
       ↪weight_decay=1e-5)
      scheduler = lr_scheduler.MultiplicativeLR(optimizer, lr_lambda=lambda _: 0.
       ↪999)

      loss_hist = train_loop(model_2m, optimizer, scheduler, moons_dl, epochs)

      plt.figure(figsize=(10, 5))
      plt.plot(loss_hist[epochs//10:], label='loss')
      plt.legend()
      plt.show()
```



Once the model is trained we can use it for sampling. We can see that the fit is not perfect, but we obtain two modalities that follow the original distribution. Notice that to obtain a sample from the original distribution we are sampling the gaussian distribution and simply transform it with the model using that the model is an invertible function.

```
[11]: samples = model_2m.sample(num_samples=2048)[0].cpu().detach().numpy()
      plt.scatter(samples[:,0], samples[:,1]);
```



We can also use as target a distribution more similar to out distribution, for instance a mixture of gausian with to modes, so the correspondence is easier.

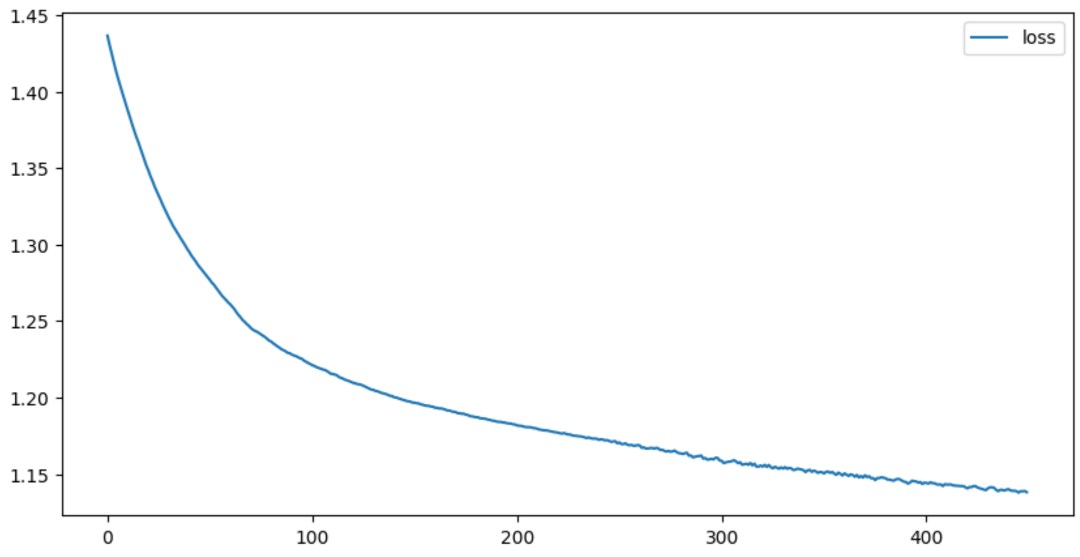
[12]:
```
# Define a bimodal gaussian mixture
baseGM = nf.distributions.base.GaussianMixture(2,2, loc=np.
 ↪array([[-2,0],[2,0]], dtype=np.float32),scale=np.array([[0.3,0.3],[0.3,0.
 ↪3]], dtype=np.float32))
```

We define again the same flow (you can check if a smaller model obtains a good result)

[13]:
```
model_2m_GM = nf.NormalizingFlow(baseGM, flows).to(device)
```

We fit the model again

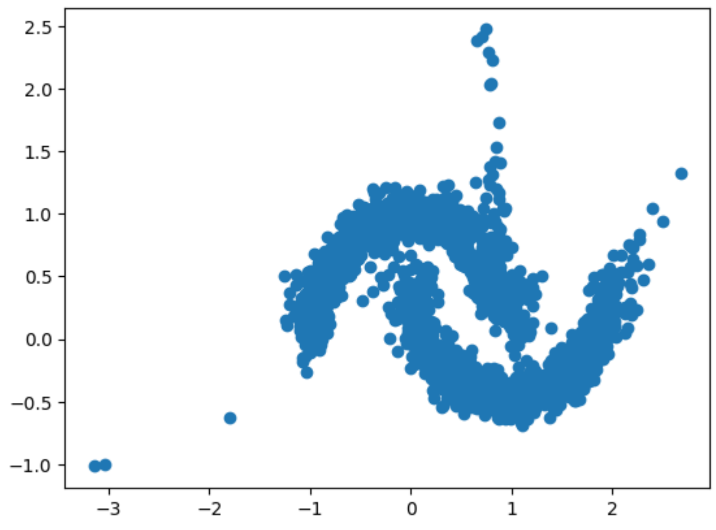
[14]:
```
optimizer = torch.optim.Adamax(model_2m_GM.parameters(), lr=1e-3,␣
 ↪weight_decay=1e-5)
scheduler = lr_scheduler.MultiplicativeLR(optimizer, lr_lambda=lambda _: 0.
 ↪999)
epochs = 500
loss_hist = train_loop(model_2m_GM, optimizer, scheduler, moons_dl, epochs)

plt.figure(figsize=(10, 5))
plt.plot(loss_hist[epochs//10:], label='loss')
plt.legend()
plt.show()
```



Sampling we can see that the fit is slightly better than with the unimodal gaussian and it converges faster.

Obviously we can increase the number of layers on the first model perform more optimization steps, but that will increase the computational cost.

[15]:
```
samples = model_2m_GM.sample(num_samples=2048)[0].cpu().detach().numpy()
plt.scatter(samples[:,0], samples[:,1]);
```



8.3.3 Flows - Digits Dataset

Now we are apply the differnt flows that have been explained to the digits dataset as in the autorregresive models.

[16]:
```
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
```

We select also the 0 digit

[17]:
```
Xo,y = load_digits(n_class=10, return_X_y=True)
X = Xo[y==0]
```

Now we model the digits as continuous varables (not binary), we normalize the data so it is in the range [-1,1]

[18]:
```
digits_dl = torch.utils.data.DataLoader(torch.tensor((X/8)-1).type(torch.
 ↪float32), batch_size=64)
```

8.3.4 Masked Autoregressive Flow

We first test the autoregresive flows. In this case we use layers that model the transformation as an autoregressive distribution based on the MADE model.

We define a base distribution that has as many dimensions as the size of the digits.

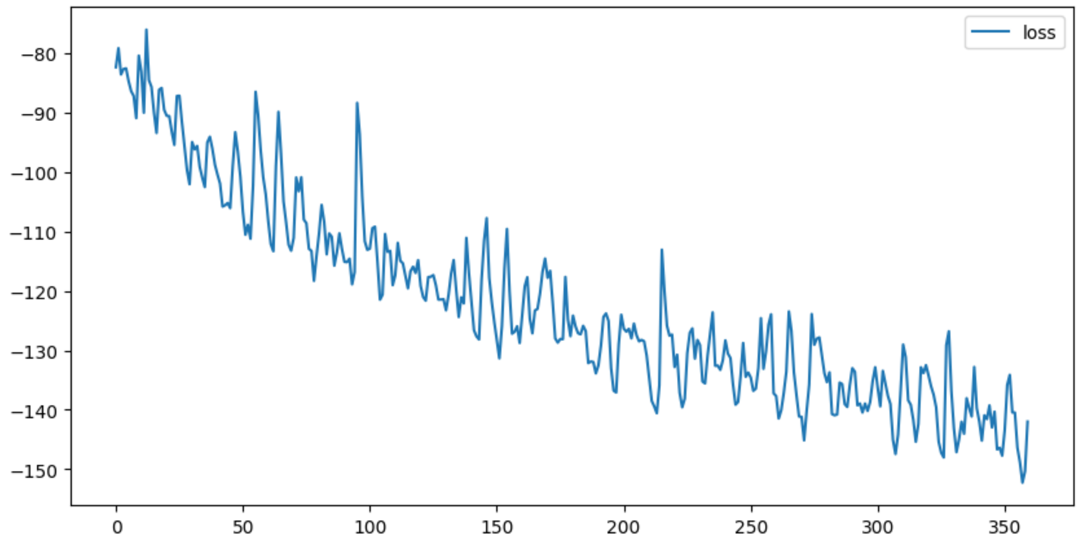
[19]:
```
base = nf.distributions.base.DiagGaussian(64)
```

We define an autoregresive flow using the MADE layers.

[20]:
```
# We define the flow layers
num_layers = 2
flows = []
for i in range(num_layers):
    # Each layer has as input the number of variables and a MADE model with␣
 ↪a certain number of
    # neurons and layers
    flows.append(nf.flows.MaskedAffineAutoregressive(features=64,␣
↪hidden_features=96, num_blocks=2))

# We define the flow from the base distribution and the flow layers
MAF = nf.NormalizingFlow(base, flows).to(device)
```

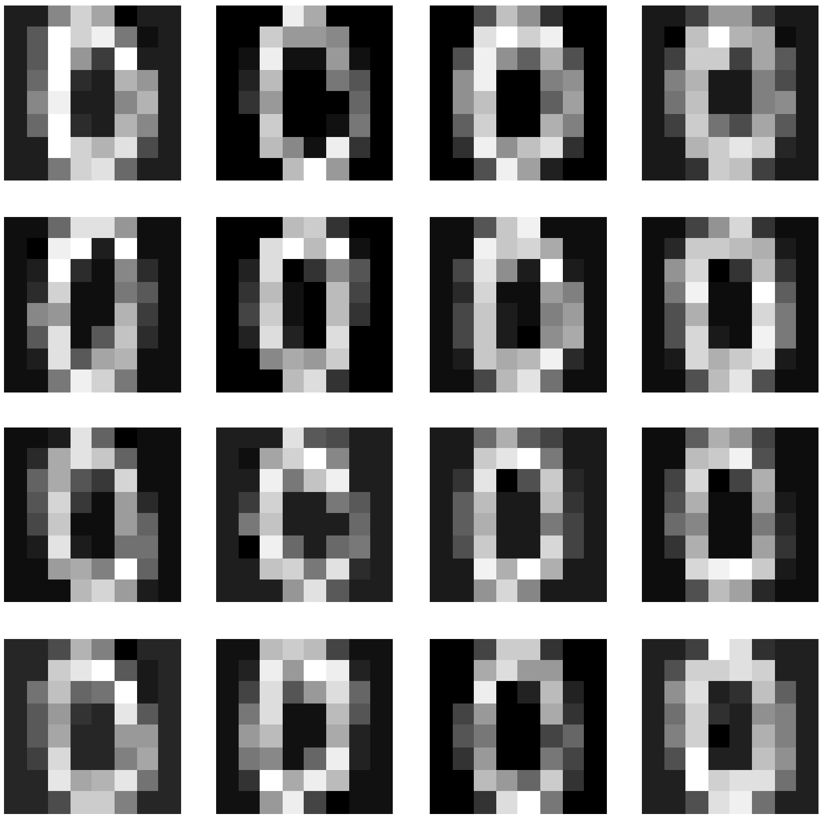
Now we fit the flow

[21]:
```
optimizer = torch.optim.Adamax(MAF.parameters(), lr=1e-2, weight_decay=1e-5)
scheduler = lr_scheduler.MultiplicativeLR(optimizer, lr_lambda=lambda _: 0.
↪9999)
epochs = 400
loss_hist = train_loop(MAF, optimizer, scheduler, digits_dl, epochs)

plt.figure(figsize=(10, 5))
plt.plot(loss_hist[epochs//10:], label='loss')
plt.legend()
plt.show()
```



Now we can sample from the model, as before, sampling a 64 dimensional gausian and transforming the sample using the inverse function.

[22]:
```
nim = 4
samples = MAF.sample(num_samples=nim*nim)[0].cpu().detach().numpy()
fig = plt.figure(figsize=(6,6))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(np.trunc((samples[i].reshape(8,8)+1)*8),cmap='gray')
    plt.axis('off')
```



The model can also compute the log likelihood in this case using the gaussian distribution

[23]:
```
MAF.log_prob(torch.tensor((X[:100]/8)-1, dtype=torch.float32).to(device)).
↪detach().sum()
```

[23]:
```
tensor(9361.2363, device='cuda:0')
```

8.3.5 Conditioned Masked Autoregressive Flow

Modelling a distribution we can input additional information that condition the generation, this way we can control in this case the examples that are generated.

We can condition to any available information. In this case we know the class labels, so we can try to generate more that one class of digit adding this information as conditioning.

We define a dataloader for the digits 2 and 6

[52]:
```
class Digits(Dataset):
    def __init__(self, data, labels):
        self.data = torch.Tensor((data/8)-1).type(torch.float32)
        self.labels = torch.Tensor(labels).type(torch.float32).unsqueeze(1)


    def __getitem__(self, index):
        return self.data[index], self.labels[index]

    def __len__(self):
        return len(self.data)
```

```
digdata = Digits(Xo[np.logical_or(y==2, y==6)], y[np.logical_or(y==2, y==6)])
twodig_dl = torch.utils.data.DataLoader(digdata, batch_size=64)
```

We use the same base distribution

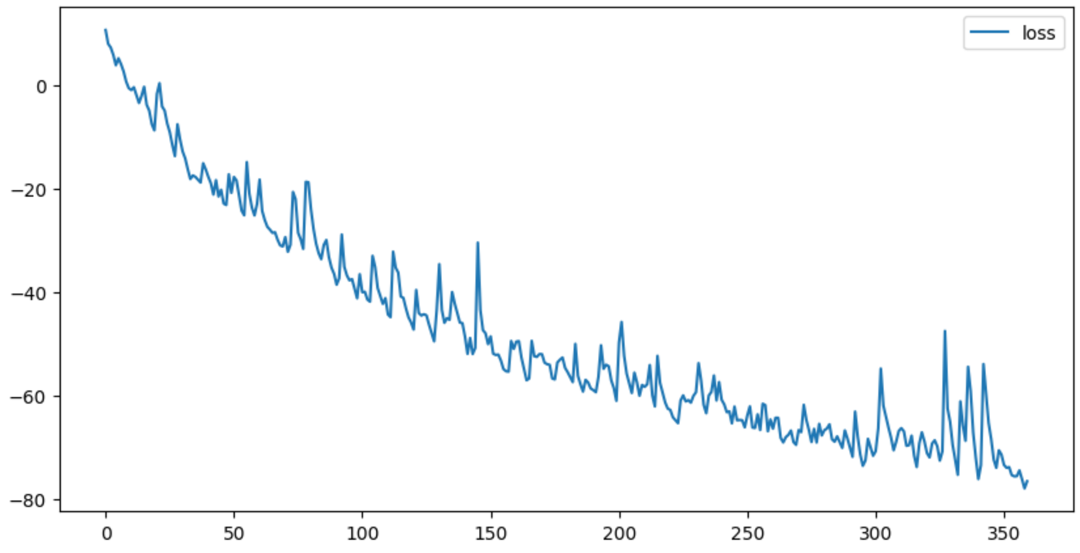
[67]:
```
basec = nf.distributions.base.DiagGaussian(64)
```

The MADE model allows introducing the charateristics used as conditioning (`context_features`). The model is defined so it applies this information to condition the probability distribution.

[68]:
```
# We definr the flow as before
num_layers = 2
flows = []
for i in range(num_layers):
    flows.append(nf.flows.MaskedAffineAutoregressive(64, 96,␣
 ↪context_features=1, num_blocks=2))

# We build the conditional flow
cMAF = nf.ConditionalNormalizingFlow(basec, flows).to(device)
```

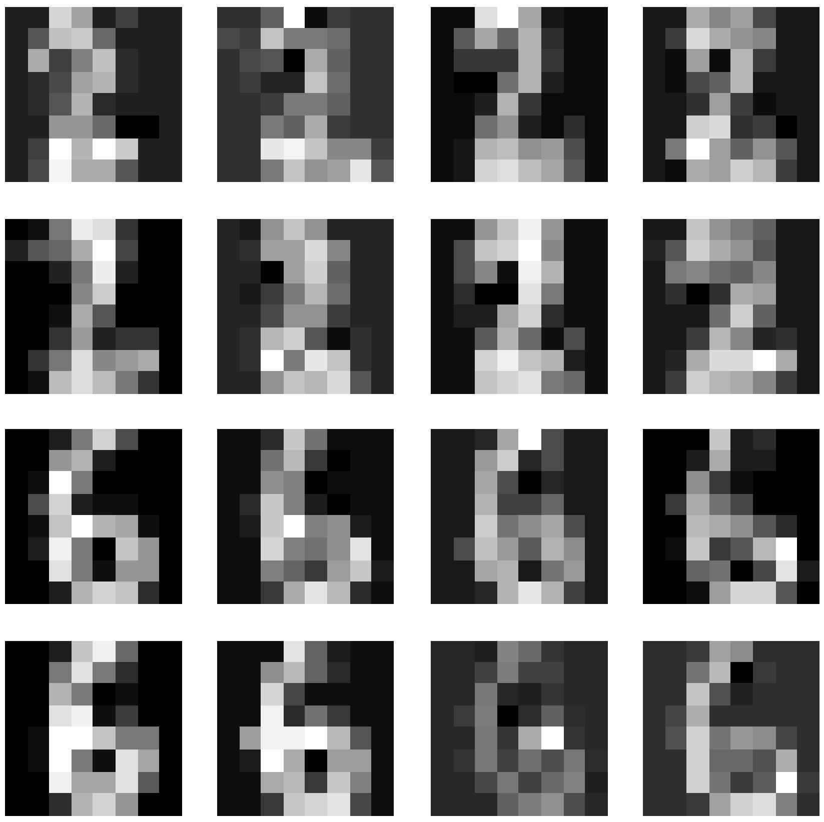
[69]:
```
epochs = 400 # This model takes longer to train in Colab, so we reduce the␣
 ↪number of epochs
```

[70]:
```
optimizer = torch.optim.Adamax(cMAF.parameters(), lr=5e-3, weight_decay=1e-5)
scheduler = lr_scheduler.MultiplicativeLR(optimizer, lr_lambda=lambda _: 0.
 ↪9999)

loss_hist = train_loop(cMAF, optimizer, scheduler, twodig_dl, epochs,␣
 ↪conditional=True)

plt.figure(figsize=(10, 5))
plt.plot(loss_hist[epochs//10:], label='loss')
plt.legend()
plt.show()
```



For sampling, we simply give to the model the conditioning that will use the flow to map from the gaussian to the corresponding digit.

[71]:
```
nim = 4
c = ([2.0] * ((nim*nim)//2))  + ([6.0] * ((nim*nim)//2))
cond = torch.tensor([c]).type(torch.float32).to(device)
samples = cMAF.sample(num_samples=nim*nim, context=cond.T)[0].cpu().detach().
 ↪numpy()
fig = plt.figure(figsize=(6,6))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(np.trunc((samples[i].reshape(8,8)+1)*8),cmap='gray')
    plt.axis('off')
```



8.3.6 Affine Coupled Flow

Other way of defining flows is using affine transformations, that is the basis of the NICE and RealNVP flows

This is more parallelizable than using an autoregressive formulation since we can apply the transformation in parallel.

The idea of using an affine transformation is that it defines a Jacobian that is triangular, so it is computationally inexpensive to do gradient descent with the loss function (log likelihood).

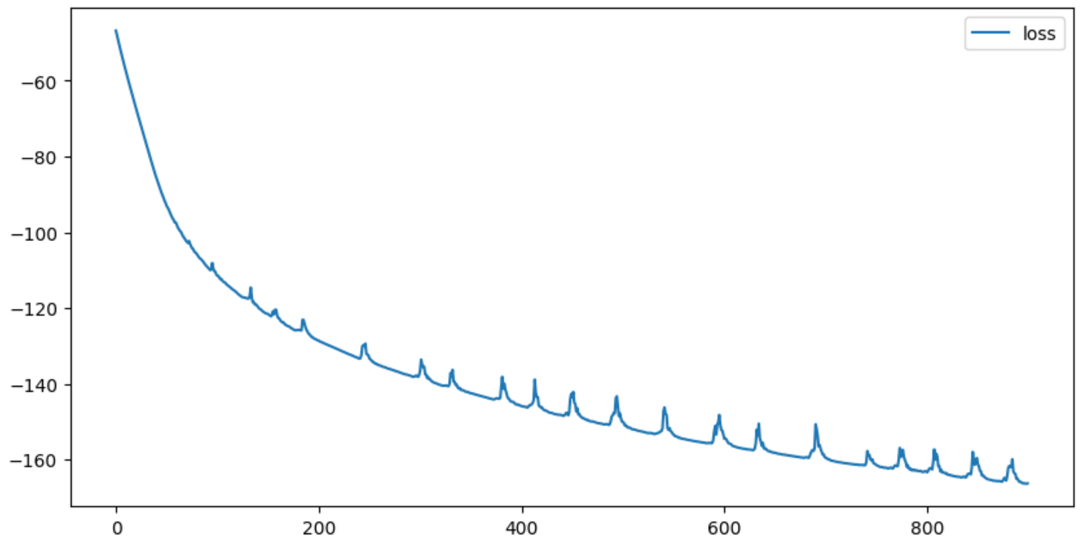
If you remember how these flows work, the idea is to chain successive layers where we transform half of the variables and leave the rest untouched, in each layer we transform a set of different variables. The transformation (scaling/translation) is calculated using a neural network.

[30]:
```
num_layers = 10
flows = []
```

```
for i in range(num_layers):
    # The MLP defines the network that models the affine transformation␣
 ↪parameters
    # translation/scaling as (mean/variance)
    param_map = nf.nets.MLP([32,64, 64, 64], init_zeros=True)
    # Affine transformation
    flows.append(nf.flows.AffineCouplingBlock(param_map))
    # Shuffle the variables
    flows.append(nf.flows.Permute(64, mode='shuffle'))

base = nf.distributions.base.DiagGaussian(64)

# Build the flow
acf = nf.NormalizingFlow(base, flows).to(device)
```

[31]:
```
epochs = 1000 # This models are faster, so we can train them for more epochs
```

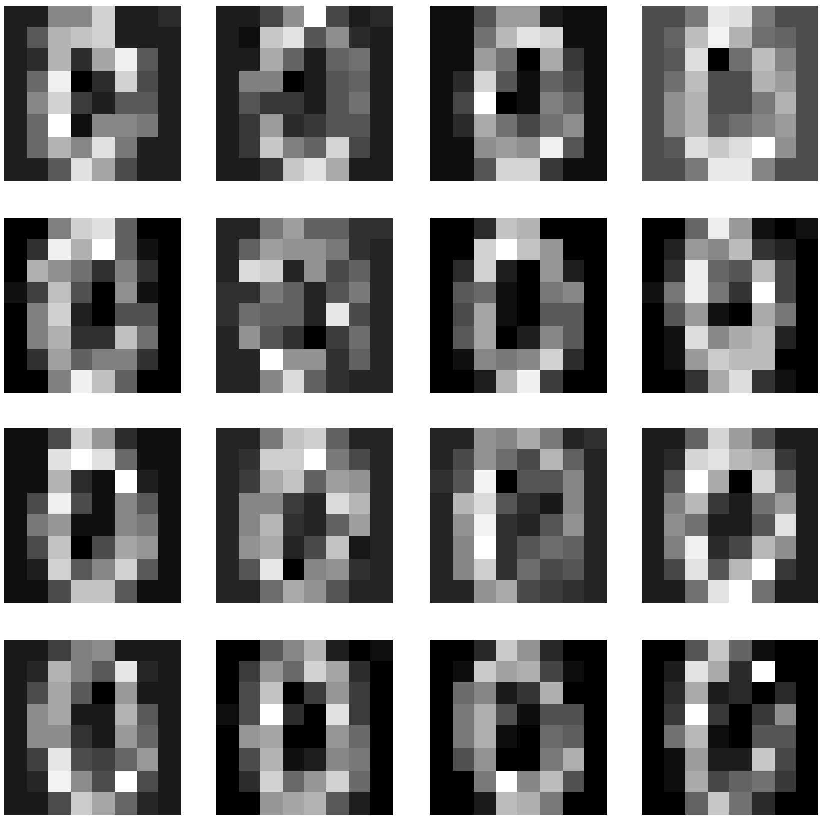
[32]:
```
optimizer = torch.optim.Adamax(acf.parameters(), lr=1e-4, weight_decay=1e-5)
scheduler = lr_scheduler.MultiplicativeLR(optimizer, lr_lambda=lambda _: 0.
 ↪999)

loss_hist = train_loop(acf, optimizer, scheduler, digits_dl, epochs)

plt.figure(figsize=(10, 5))
plt.plot(loss_hist[epochs//10:], label='loss')
plt.legend()
plt.show()
```



Hyperparameters have been manualy adjusted, feel free to explore them further

[33]:
```
nim = 4
samples = acf.sample(num_samples=nim*nim)[0].cpu().detach().numpy()
fig = plt.figure(figsize=(6,6))
```

```
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(np.trunc((samples[i].reshape(8,8)+1)*8),cmap='gray')
    plt.axis('off')
```



[34]:
```
acf.log_prob(torch.tensor((X[:100]/8)-1, dtype=torch.float32).to(device)).
 ↪detach().sum()
```

[34]:
```
tensor(11125.9375, device='cuda:0')
```

8.3.7 Real NVP

We can obtain more complex flows by making changes to how the transformations are performed.

As we have seen, Real NVP uses an affine transformation based on scaling and translation. In this case we calculate them independently with an MLP and use masking instead of permutations. The masking used alternates the use of variables in consecutive order (in 2D it is a checkerboard) so that we preserve some locality in each transformation layer. The normalization of activations added after each transformation should help stabilize the training (or not)

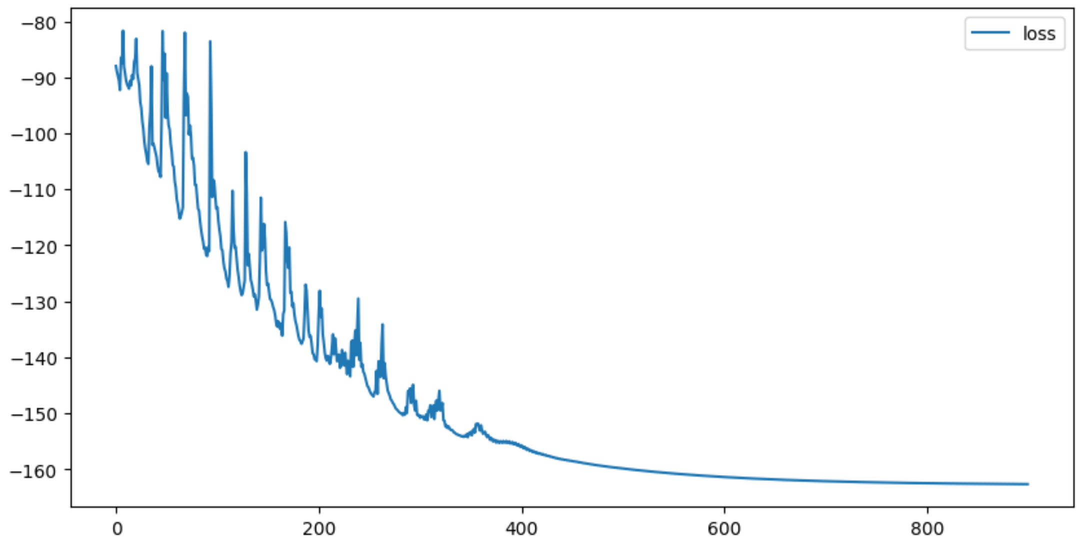
[35]:
```
K = 8
latent_size = 64
b = torch.Tensor([1 if i % 2 == 0 else 0 for i in range(latent_size)])
flows = []
```

```
for i in range(K):
    # Network to calculate scaling and translation
    s = nf.nets.MLP([latent_size, 2 * latent_size, 2 *␣
 ↪latent_size,latent_size], init_zeros=True)
    t = nf.nets.MLP([latent_size, 2 * latent_size, 2 *␣
 ↪latent_size,latent_size], init_zeros=True)
    # Alternating masking for even and odd layers
    if i % 2 == 0:
        flows += [nf.flows.MaskedAffineFlow(b, t, s)]
    else:
        flows += [nf.flows.MaskedAffineFlow(1 - b, t, s)]
    # Activation normalization to stabilize training
    flows += [nf.flows.ActNorm(latent_size)]

rnvp = nf.NormalizingFlow(base,flows).to(device)

# Init activation normalization by generating some samples
z, _ = rnvp.sample(num_samples=2 ** 7)
```

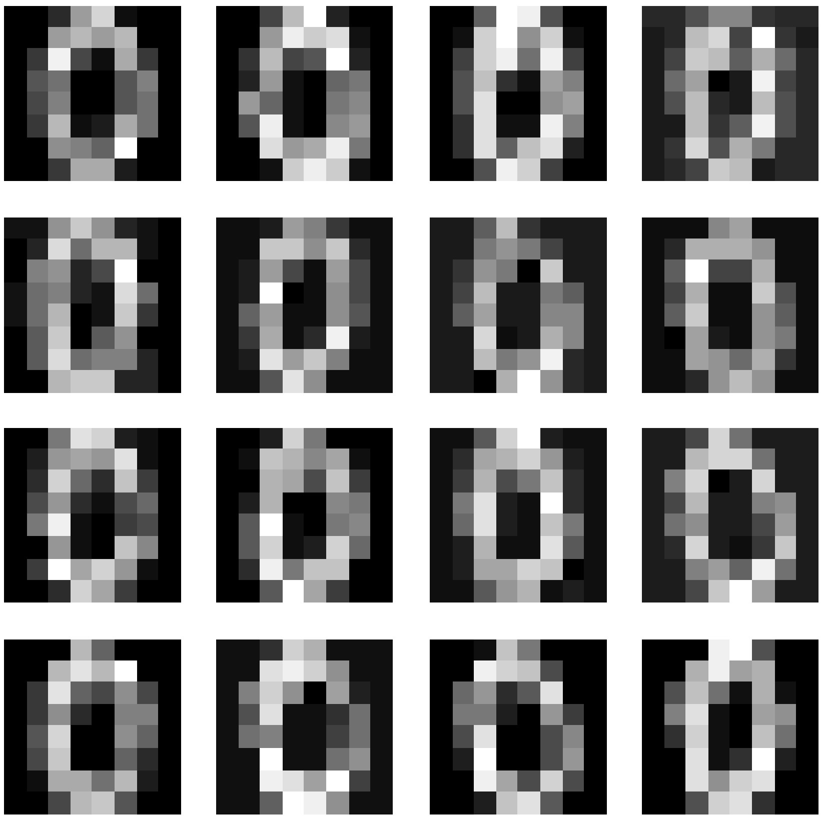
Hyper parameters adjusted again manualy

[36]:
```
epochs = 1000 # Adjust if is too slow
```

[37]:
```
optimizer = torch.optim.Adamax(rnvp.parameters(), lr=1e-2, weight_decay=1e-5)
scheduler = lr_scheduler.MultiplicativeLR(optimizer, lr_lambda=lambda _: 0.
 ↪99)

loss_hist = train_loop(rnvp, optimizer, scheduler, digits_dl, epochs)

plt.figure(figsize=(10, 5))
plt.plot(loss_hist[epochs//10:], label='loss')
plt.legend()
plt.show()
```



[38]:
```
nim = 4
samples = rnvp.sample(num_samples=nim*nim)[0].cpu().detach().numpy()
fig = plt.figure(figsize=(6,6))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(np.trunc((samples[i].reshape(8,8)+1)*8),cmap='gray')
    plt.axis('off')
```



The log likelihood should have been improved

[39]:
```
rnvp.log_prob(torch.tensor((X[:100]/8)-1, dtype=torch.float32).to(device)).
↪detach().sum()
```

[39]:
```
tensor(10568.9287, device='cuda:0')
```

8.3.8 The whole enchilada

Now we are going to use a flow that has been trained with the full data set.

The following cells are for training it, but it may take around 15 minutes (probably longer in Colab), so we'll skip to the cell where we load the pre-trained model.

[40]:
```
digits_dl2 = torch.utils.data.DataLoader(torch.tensor((Xo/8)-1).type(torch.
↪float32), batch_size=128)
```

It is the same network as before but increasing the number of parameters

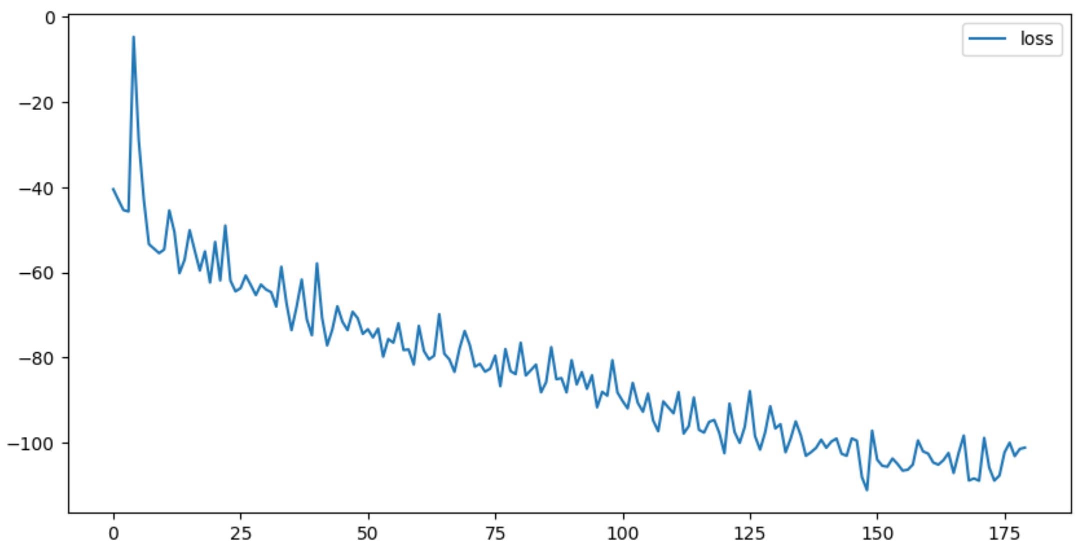
[41]:
```
K = 14
latent_size = 64
```

```
b = torch.Tensor([1 if i % 2 == 0 else 0 for i in range(latent_size)])
flows = []
for i in range(K):
    s = nf.nets.MLP([latent_size, 2 * latent_size, 2 *␣
 ↪latent_size,latent_size], init_zeros=True)
    t = nf.nets.MLP([latent_size, 2 * latent_size, 2 *␣
 ↪latent_size,latent_size], init_zeros=True)
    if i % 2 == 0:
        flows += [nf.flows.MaskedAffineFlow(b, t, s)]
    else:
        flows += [nf.flows.MaskedAffineFlow(1 - b, t, s)]
    flows += [nf.flows.ActNorm(latent_size)]

rnvp = nf.NormalizingFlow(base,flows).to(device)

z, _ = rnvp.sample(num_samples=2 ** 7)
```

Manualy adjuster

[42]:
```
epochs = 200 # We train a little bit so we can see that it works and then we␣
 ↪load the pretrained model
```

[43]:
```
optimizer = torch.optim.Adamax(rnvp.parameters(), lr=1e-2, weight_decay=1e-5)
scheduler = lr_scheduler.MultiplicativeLR(optimizer, lr_lambda=lambda _: 0.
 ↪999)

loss_hist = train_loop(rnvp, optimizer, scheduler, digits_dl2, epochs)

plt.figure(figsize=(10, 5))
plt.plot(loss_hist[epochs//10:], label='loss')
plt.legend()
plt.show()
```



[44]:
```
# Cambiar si entrenamos lo suficiente
save_model = False
if save_model:
    torch.save(rnvp.state_dict(), '/content/rnvp-digits.pt' )
```

We will continue from here

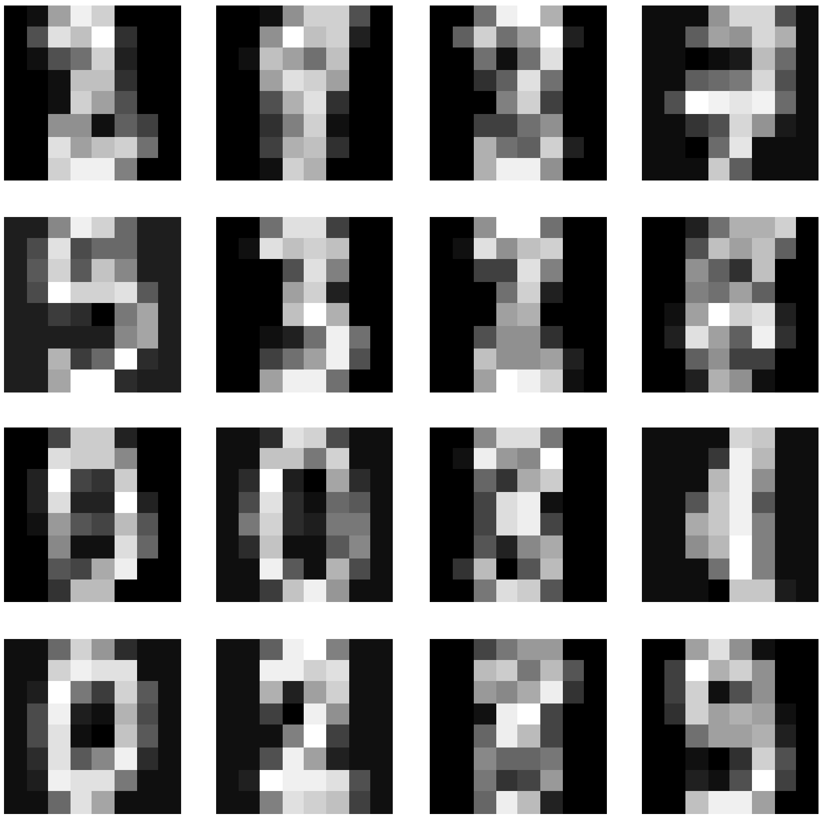
We should have uploaded to Colab the pre-trained model that is with the session material to the directory `/content`

[45]:
```
lrnvp = nf.NormalizingFlow(base,flows)
lrnvp.load_state_dict(torch.load('/content/rnvp-digits.pt'))
```

[45]:
```
<All keys matched successfully>
```

If we sample we will see that we can generate any of the 10 digits

[46]:
```
nim = 4
samples = lrnvp.sample(num_samples=nim*nim)[0].cpu().detach().numpy()
fig = plt.figure(figsize=(6,6))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(np.trunc((samples[i].reshape(8,8)+1)*8),cmap='gray')
    plt.axis('off')
```



An interesting thing about flows is that a traversal of the reference distribution (Gaussian) helps us traverse the space of the data set. In particular, we can interpolate between two examples, obtaining the corresponding points in the Gaussian and observing the transition between them.

We can chose a couple of examples in which two **different** digits are recognizable and assign their position to the variables of the next cell (the position of the upper left corner is zero)

[47]:
```
d1=0
d2=11
```

We use the inverse transformation of the flow to obtain the points that correspond to the samples in the Gaussians.

[48]:
```
start = rnvp.inverse(torch.tensor(samples[d1]).to(device))
end = rnvp.inverse(torch.tensor(samples[d2]).to(device))
```

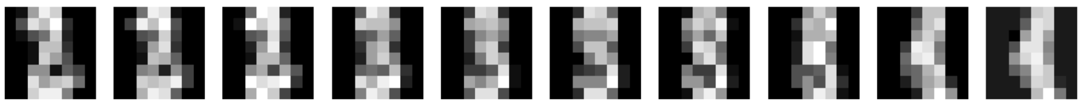
We calculate equidistant points within the vector that connects those two points within the Gaussian distribution

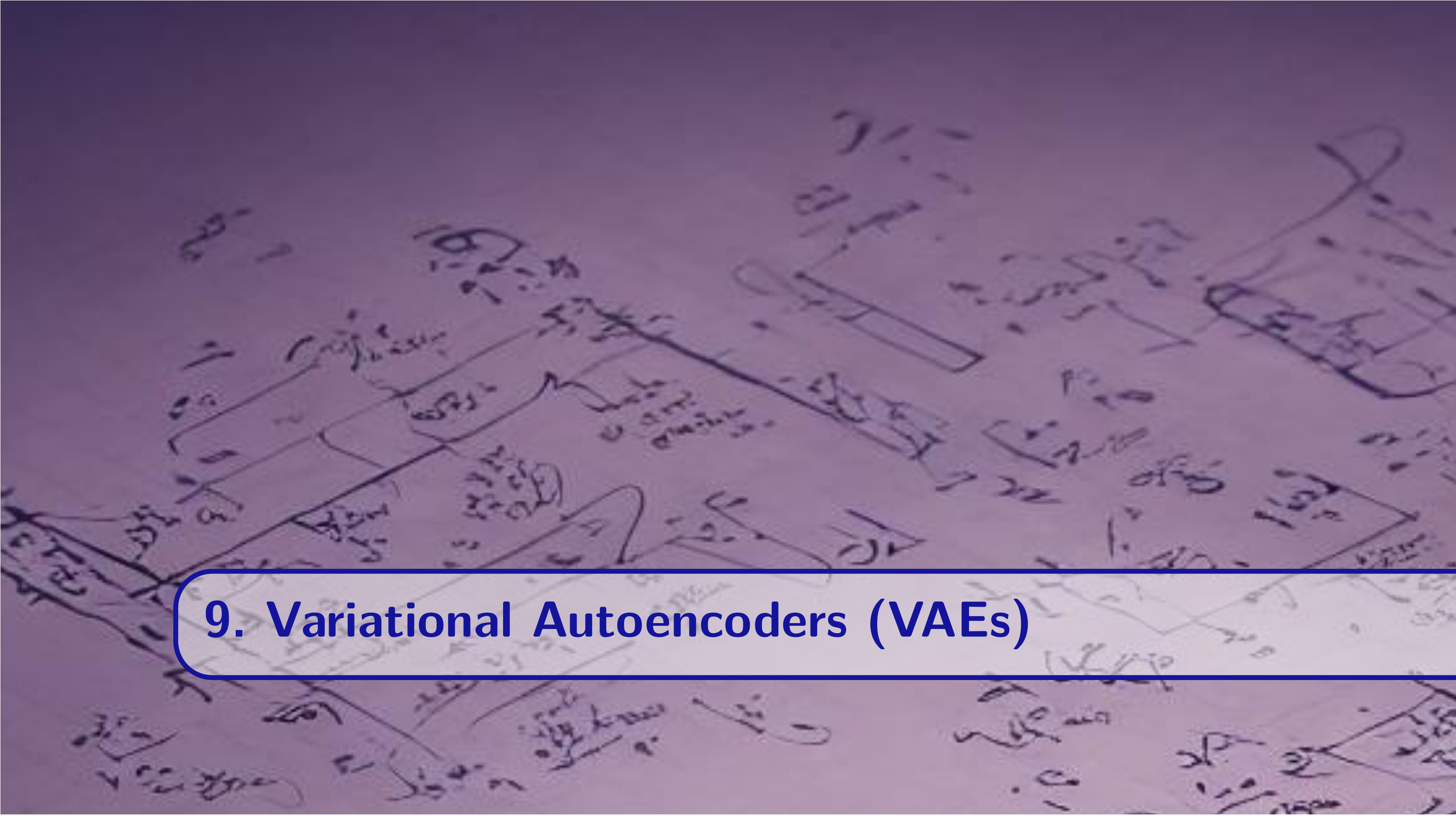
[49]:
```
interp = []

for v in torch.arange(0,1,0.1):
    interp.append(torch.lerp(start,end, v.to(device)))
```

Now we can see what lies in the path between the two samples simply by applying the flow to the points on the Gaussian.

Depending on how the correspondence between the digits and the Gaussian has been made and where the digits are located, other intermediate digits may appear.

[50]:
```
fig = plt.figure(figsize=(15,10))
for i, v in enumerate(interp):
    plt.subplot(1, len(interp), i+1)
    plt.imshow(np.trunc((rnvp(v).cpu().detach().numpy().
 ↪reshape(8,8)+1)*8),cmap='gray')
    plt.axis('off')
```





9. Variational Autoencoders (VAEs)

9.1 Latent variables models

One of the weak points of autoregressive models and flows is that we have to work with models that use directly all the variables on the estimation. This constraints to networks that have a number of neurons directly proportional on the size of the problem.

Latent variable models assume that there is a set of hidden variables that generate the observed variables. This is plausible in most problems. For instance, we represent images using an array of pixels, this is for us the observable variables, but we can describe the content of the image in a more simplified way according to the elements that are in the image, like for instance, trees, grass, a horse, a person riding the horse. This is how we internally represent the content, not with the pixels, we can try to model information this way.

We want to have a set of hidden variables, probably (hopefully) less than the observable variables that are more interpretable and that hold some semantic information. They correspond to the hidden representation of the data.

The goal of latent variable models is to identify that hidden representation from the observed data. A probabilistic relationship is defined among the latent and observed variables as:

$$\begin{tikzcd} z \arrow[r, "p_\theta(x|z)"] & x \end{tikzcd}$$

This means that sampling is performed first obtaining values from the latent variables and then sampling the observable variables from a distribution that is conditioned to the values of the latents:

$$z \sim p_Z(z),\, x \sim p_\theta(x|z)$$

This allows to compute the probability (and likelihood) of the observed variables marginalizing over the latent variables given that $p(x,z) = p(x|z)p(z)$:

$$p_\theta(x) = \sum_z p_\theta(x|z)p_Z(z)$$

This makes possible to apply gradient descent for maximum likelihood of a model, defining the maximization problem:

$$\max_\theta \sum_i \log p_\theta(x_i) = \max_\theta \sum_i \log \sum_z p_\theta(x_i|z) p_Z(z)$$

To compute this model we need a probability distribution that allows finding the latent variables given the data, if we use the Bayes formula we have that

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)}$$

This leaves us with some problems, the first one is obtaining the normalization constant $p(x)$. The second one appears if we assume that z is a set of continuous variables since we need to compute:

$$\int_z p(x|z)p(z)dz$$

That needs multiple integration, that in general, for a complex distribution, is usually intractable. The usual approach to circumvent these problems is to use *Variational Inference*.

9.2 Variational Inference

The idea of variational inference is to substitute an intractable probability distribution p by another distribution q that has better properties and that can be computed more easily. The goal is to transform the distribution q tuning its parameters (ϕ) so it is as close as possible to p. At least we want that the samples from q are close locally to the p distribution. Basically, we need a surrogate tractable distribution q_ϕ that can be adapted, so it can get close to the one that we can not compute p_θ.

We can measure how close two probability distributions are in different way. A common measure is the Kullback-Leibler divergence (KL) that is defined as:

$$KL(q||p) = -\sum_x q(x) \log \frac{p(x)}{q(x)} = \sum_x q(x) \log \frac{q(x)}{p(x)}$$

This is not a proper distance, given that it does not hold the symmetrical property, but we are going to be interested on getting close only in one direction. There are other alternatives, but this one is simpler to compute using gradient descent.

So, in this problem we are going to pick a distribution $q_\phi(z|x)$ that is easy to compute as a substitute of $p_\theta(z|x)$. The parameters are going to be computed minimizing the KL divergence between the distributions. This will be our optimization objective:

$$KL(q_\phi(z|x)||p_\theta(z|x)) = \sum_{z \sim q_\phi(z|x)} q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(z|x)}$$

We are going first to operate with this objective to understand what we need to optimize. We start with:

$$KL(q_\phi(z|x)||p_\theta(z|x)) = \sum_{z \sim q_\phi(z|x)} q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(z|x)}$$

if we apply the product rule we obtain:

$$\sum_{z\sim q_\phi(z|x)} q_\phi(z|x)\log\frac{q_\phi(z|x)}{p_\theta(x,z)/p_\theta(x)} = \sum_{z\sim q_\phi(z|x)} q_\phi(z|x)\left[\log\frac{q_\phi(z|x)}{p_\theta(x,z)}p_\theta(x)\right]$$

given that the logarithm of the product is the sum of the logarithms:

$$\sum_{z\sim q_\phi(z|x)} q_\phi(z|x)\left[\log\frac{q_\phi(z|x)}{p_\theta(x,z)}+\log p_\theta(x)\right]$$

we can now apply the distributive:

$$\sum_{z\sim q_\phi(z|x)} q_\phi(z|x)\log\frac{q_\phi(z|x)}{p_\theta(x,z)} + \sum_{z\sim q_\phi(z|x)} q_\phi(z|x)\log p_\theta(x)$$

on the second term the sum does not depend on x:

$$\sum_{z\sim q_\phi(z|x)} q_\phi(z|x)\log\frac{q_\phi(z|x)}{p_\theta(x,z)} + \log p_\theta(x)\sum_{z\sim q_\phi(z|x)} q_\phi(z|x)$$

on the second term we sum over all $q_\phi(z|x)$, so it sums to 1, so we finish with

$$KL(q_\phi(z|x)||p_\theta(z|x)) = \sum_{z\sim q_\phi(z|x)} q_\phi(z|x)\log\frac{q_\phi(z|x)}{p_\theta(x,z)} + \log p_\theta(x)$$

we can finally reorder the terms to obtain the expression:

$$\begin{aligned}\log p_\theta(x) &= KL(q_\phi(z|x)||p_\theta(z|x)) - \sum_{z\sim q_\phi(z|x)} q_\phi(z|x)\log\frac{q_\phi(z|x)}{p_\theta(x,z)}\\ &= KL(q_\phi(z|x)||p_\theta(z|x)) + \sum_{z\sim q_\phi(z|x)} q_\phi(z|x)\log\frac{p_\theta(x,z)}{q_\phi(z|x)}\end{aligned}$$

Where $\log p_\theta(x)$ is the log likelihood of the distribution of the data. The second term of the last expression is called the Variational Lower Bound (VLB) or the Evidence Lower Bound (ELBO). Since the KL divergence is always equal or larger than zero, the ELBO is a lower bound of the probability distribution $p(x)$. When $KL(q_\phi(z|x)||p_\theta(z|x))$ is exactly zero, the ELBO corresponds to the likelihood of the p distribution. So minimizing the divergence between the two distributions, we are maximizing the likelihood of the data.

The KL divergence is actually defining two distances, how close the two distributions are and what is the separation (tightness) between the ELBO and the log likelihood of the marginal distribution.

We can also rewrite the ELBO to obtain an alternative interpretation, operating we obtain:

$$\begin{aligned}\sum_z q_\phi(z|x)\log\frac{p_\theta(x,z)}{q_\phi(z|x)} &= \sum_z q_\phi(z|x)\log\frac{p_\theta(x|z)p_\theta(z)}{q_\phi(z|x)}\\ &= \sum_z q_\phi(z|x)\log p_\theta(x|z)\frac{p_\theta(z)}{q_\phi(z|x)}\\ &= \sum_z q_\phi(z|x)\left[\log p_\theta(x|z)+\log\frac{p_\theta(z)}{q_\phi(z|x)}\right]\\ &= \sum_z q_\phi(z|x)\log p_\theta(x|z) + \sum_z q_\phi(z|x)\log\frac{p_\theta(z)}{q_\phi(z|x)}\\ &= \sum_z q_\phi(z|x)\log p_\theta(x|z) - KL(q_\phi(z|x)||p_\theta(z))\end{aligned}$$

Where the first term corresponds to a reconstruction term that maintains the correspondence between observed and latent variables and the second corresponds to a regularization term, that maintains the distribution $p_\phi(z|x)$, the one that defines the link from the observed variables to the latent variables, close to the prior distribution chosen for the latent variables.

The variational lower bound can be defined as expectations (mean over samples) and this can be optimized using gradient descent using samples from the distributions. We can rewrite the equality as:

$$\log p_\theta(x) = \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{q_\phi(z|x)}{p_\theta(z|x)}\right] + \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}\right]$$

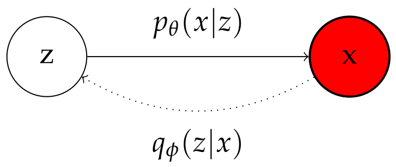
Now we have to worry only about to maximize the ELBO, since doing that we maximize the log likelihood of the marginal distribution $p(x)$ and we minimize the divergence between the distributions $q_\phi(z|x)$ and $p_\theta(z|x)$. This give us the loss:

$$\mathcal{L}_{\theta,\phi}(x) = \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}\right] = \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x,z) - \log q_\phi(z|x)\right]$$

This is a two criteria optimization function, fitting the parameters ϕ and θ can be done by iterative optimization using alternate gradient descent. This means to perform two steps for each iteration, first computing the derivative of one group of parameters, fixing the other and then changing the roles.

Gradient descent will work with samples and the derivatives that can be computed from the function that defines the ELBO. This function ($\mathcal{L}_{\theta,\phi}(x)$) is not always tractable, but the derivative respect to θ can be computed from the derivative of the function that computes p. The derivative respect to ϕ will depend on our choice for the distribution q. We will pick a tractable distribution. As we will see soon we will need some tricks to be able to use this with a neural network.

In the end, the goal of all these estimations and procedure is to have probability distributions that can map from the original data space to the latent space and back



For the structure of this mapping we can see that it can be implemented as a type of autoencoder. The encoder maps samples of the real data to samples from a probability distribution of our choice and the decoder maps from samples of that chosen probability distribution to the real data. The training will aim to reconstruct the data passing through the transformation. This kind of autoencoder to no surprise is called Variational Auto Encoder (VAE).

9.3 Variational Autoencoders (VAEs)

As we saw on the first chapter of this notes (see 1.3.2), an autoencoder is a fully connected network that reproduces the input in the output, by minimizing the reconstruction error and mapping the data to an intermediate space. The *encoder* transforms the input into a low dimensional embedding using successive layers that reduce progressively the number of neurons to transform the data to a low dimensional space compressing its information. The *Decoder* transforms the code again into the original input applying successively layers that increase the

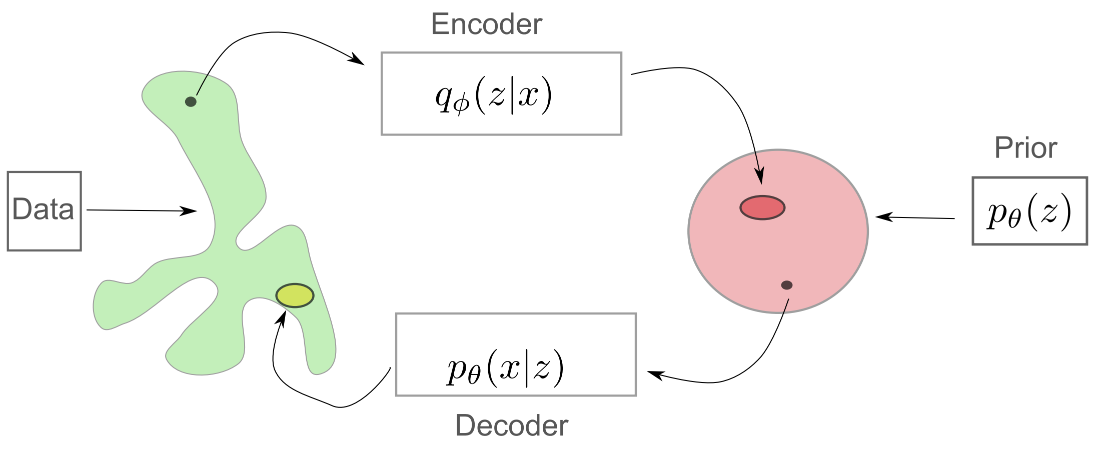


Figure 9.1: Mapping from the data to the latent space using a VAE

number of neurons. The encoder and decoder are symmetrical, so we have a mapping and an inverse mapping from the data to the intermediate encoding. An autoencoder is deterministic, meaning that for each sample we obtain directly a code and a code will always obtain the same result. We need a probabilistic autoencoder, so it can be sampled.

As we saw on the chapter of flows, we can train a network that predicts the parameters of the probability distribution we want to estimate, this is the approach we are going to use, and then we can use the these parameters for sampling from the distribution.

Hence, a Variational Auto Encoder will be composed by two networks as the normal auto encoder. The encoder learns the function $q_\phi(z|x)$ by obtaining an estimate for its parameters using samples. This network is also called the *inference model* or *recognition network*. From these parameters estimates we can obtain random samples that we will use to learn the decoder. The training will try to tune the parameters so the samples that we will draw get close to the real data. The decoder learns the function $p_\theta(x|z)$ using the samples obtained from the encoder. That will correspond to our generative model, but in this case the source distribution has fewer dimensions than the data.

The training will make that the pair encoder-decoder gets close to the distribution of the data basically by maximizing their likelihood and matching the samples in the input. Figure 9.1 represents this transformation from the data to a latent space using the encoder and decoder networks.

After training, we only need the decoder. We can obtain samples just by processing samples from a prior distribution $p_\theta(z)$.

9.4 Gaussian Variational Auto Encoder

In order to train a Variational Auto Encoder we need to decide what tractable distribution q we are going to fit in the latent space. This has to be a distribution that is simple to wok with. As usual, the Gausssian distribution is a good choice for continuous variables. So, we are going to have:

$$q_\phi(z|x) = \mathcal{N}(z;\mu,\Sigma)$$

Also, to make easier the fitting we are going to assume that all the variables are independent (diagonal covariance matrix). This will reduce the number of parameters that will have to be estimated to a linear function of the dimensionality of the latent space. This means that we will

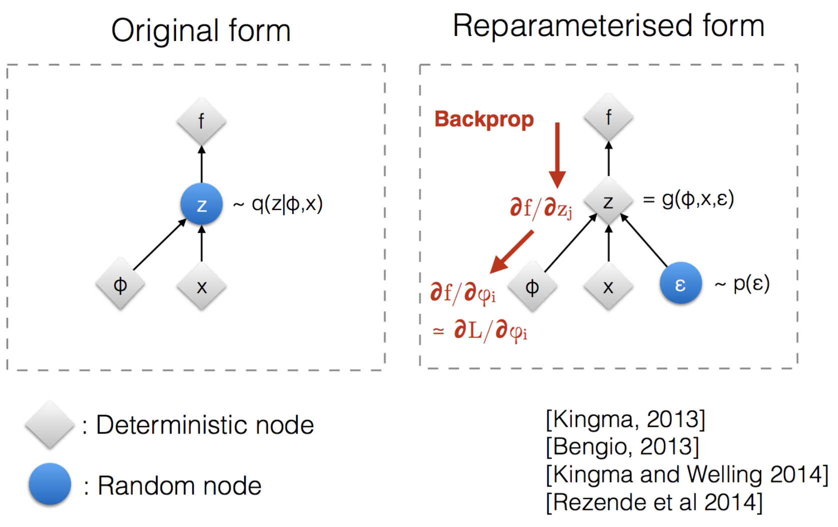


Figure 9.2: Reparameterization trick for backpropagating through sampling

fit the distribution:

$$q_\phi(z|x) = \mathcal{N}(z, \mu, diag(\sigma)) = \prod_i q_\phi(z_i|x) = \prod_i \mathcal{N}(z_i; \mu, \sigma_i)$$

The encoder only will have to estimate the μ and σ of each gaussian variables. A usual choice when performing the estimation is to fit $\log(\sigma)$ instead of directly σ for numerical stability. Once we have the parameters we can sample from the joint gaussian distribution and use the samples for the decoder.

This last steps actually represents a problem for training neural networks with backpropagation, since sampling is a probabilistic operation and can not be used with it. For solving this problem the approach is to use what is called the **reparameterization trick**. This means to define our target distribution through another by transforming the samples, so they correspond to the target distribution. In this case we can choose as base distribution unitary gaussian noise ($\epsilon \sim \mathcal{N}(0,1)$) that provides the samples in the forward step that are scaled and translated using the parameters of the target distribution:

$$z = \mu + \sigma \odot \epsilon$$

This can be forward propagated through the decoder and when performing backpropagation we can ignore the sampling node since it does not have to be changed by it. Figure 9.2 shows graphically how the computation is performed. We can see that the sampling node ϵ is left out from the backpropagation on the right side of the graphic.

Figure 9.3 summarizes all the elements of the gaussian VAE architecture. The encoder learns from the data the parameters for the Gaussian distribution that is our variational distribution. This allows us to obtain samples for the distribution of the latent variables that can then be decoded to obtain again the input sample.

We can use *conditional sampling* in VAEs as in the previous models. Since we are using neural networks to implement the probability distributions, we have additional inputs in the encoder and the decoder that will correspond to the conditioning and will bias the sampling.

A simple way is to include this information on the input layer of both networks, so it is used to estimate the distribution parameters on the encoder and to change the computation of the decoder. Other option is to use this additional information as layer normalizations as we will

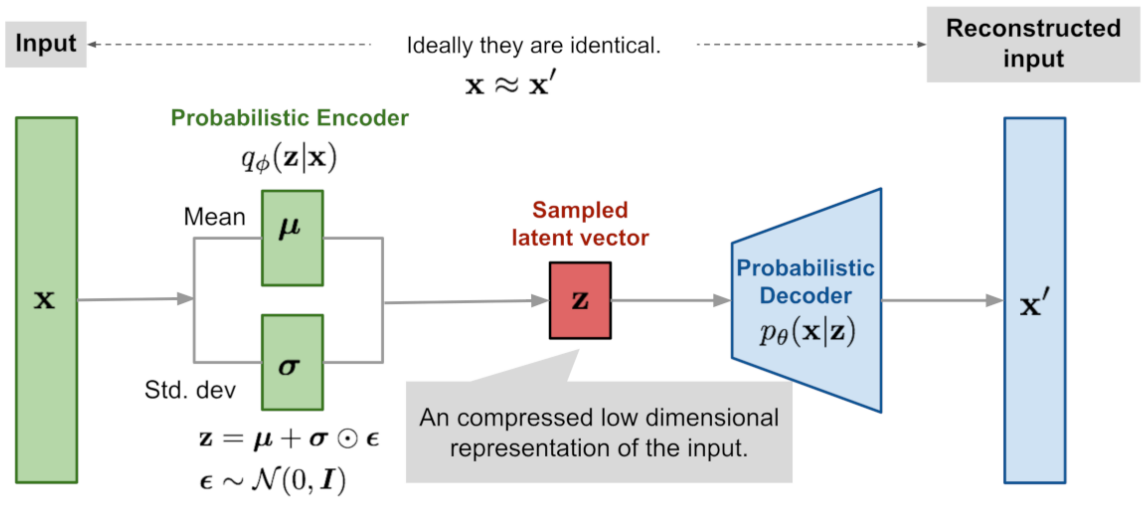

Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
Probabilistic Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
$\mathbf{x}$
Mean
$\boldsymbol{\mu}$
Std. dev
$\boldsymbol{\sigma}$
$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$
Sampled latent vector
$\mathbf{z}$
An compressed low dimensional representation of the input.
Probabilistic Decoder
$p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$

Figure 9.3: Architecture of the Gaussian Variational Autoencoder

see in other models. In this case have additional learned parameters on certain layers that will be used to scale and translate the activations accordingly to the conditioning.

9.4.1 Sampling VAEs

To assume independent gaussian distributions as priors for the latent variables is a simple approach, but could end in problems when sampling from the decoder. It is possible than the actual latent variables follow more complex distribution so, there is no correspondence with the prior that has been fitted. This can be observed when obtaining samples, it could happen that high density areas for the prior distribution do not correspond to high density areas of the real data, so we end obtaining rare samples very frequently. The opposite can also happen, we can have high density areas of the real data mapped to low probability areas of the latent priors, so these samples appear with low frequency.

There are different approaches that can be used to reduce this problem:

1. We could use a prior distribution for the latents that is more complex than the gaussian distribution, for instance a mixture of gaussians.

2. We can use a flow that maps the samples we obtain from the encoder (that are gaussian) to a more complex distribution that will adapt better to the real data. In this case the reconstruction loss will help to fit the flow to a distribution that matches the real data and the flow will bridge from the gaussian to that distribution.

3. We could use a hierarchical VAE. This kind of VAEs assume that we have a multi-stage transformation. Each step will transform a set of latent variables to a more simple (lower dimensional) set of latents. This will reduce the mismatch between the successive sets of gaussian latent and the real data.

4. We can train a model a posteriori that captures the actual distribution of the latents respect to the real data. For instance, we can encode new samples and fit a gaussian mixture model that represents better the actual densities of the encoded data, then we can use the gaussian mixture to sample latents. More complex models can also be uses, for instance autoregressive models as we will see on the last section of the chapter.

9.5 VAEs and Disentanglement

One of the aims of working with a latent space is to obtain a set of variables that have a semantic interpretation that can be used in different ways, like for interpretability or for manipulating the data according to the meaning of the variables.

The goal is to disentangle the variables that represent the original data, so they correspond to specific characteristics of that data. For instance, in the domain of images we could be interested on having variables that change the position of the elements on the image or the illumination. For problems in specific domains, like faces, we could want variables that can control the age, gender, the expression or other characteristics of the face.

It could be interesting to be able to separate the latent representation obtained from VAEs into meaningful features. When we explained an alternative formulation of the ELBO, we define its expression as:

$$ELBO = \sum_z q_\phi(z|x) \log p_\theta(x|z) - KL(q_\phi(z|x)||p_\theta(z))$$

And we identified a reconstruction term and a regularization term. The regularization term control how close is the variational distribution $q_\phi(z|x)$ to the prior distribution that we are considering for the latent variables $p_\theta(z)$. In our case unit independent gaussians. So, if we give more weight to that term, we will force the optimization to obtain more independent latent variables.

β-VAE [63] redefines the loss of the autoencoder as:

$$\mathcal{L}_{\theta,\phi}(x) = \mathbb{E}_{z\sim q_\phi(z|x)}[\log p_\theta(x|z)] - \beta KL(q_\phi(z|x)||p_\theta(z))$$

The larger the parameter β the more we force the similarity among the two distributions, with $\beta = 1$ we have the original variational autoencoder.

There is no guarantee that all the latent variables correspond to a meaningful direction of change, but depending on the data some variables will have an identifiable effect. This will need to be tested by obtaining samples from latent vectors and then changing the values of the variables on the latent vectors using orthogonal directions.

9.6 State of the art VAEs

There are some VAEs capable of learning good representation from images. The more interesting ones are VQ-VAE [104] and its sequel VQ-VAE-2 [111].

VQ-VAE uses vector quantization to obtain a discrete representation that is used for learning a set of discrete variables. Vector quantization is a method for mapping a set of K-dimensional vectors to a finite code, that is, a finite dictionary that maps continuous data to a discrete representation. The discrete code is associated with a set of vector centroids that are the ones that allow translating the continuous vectors to symbols from the dictionary. In this case the variational distribution q (the latent variables) is a discrete distribution. This discretization strategy is very useful and has been used with other generative models for obtaining more simple representation that can help to scale computation in domains where the input has many dimensions.

The goal of VQ-VAE is to learn the dictionary and the mapping that allows a good reconstruction of the input. The architecture is composed by an encoder, that obtains an embedding for the input, a quantizer that discretizes the vectors from the embedding using the codebook of K vectors that defines a discrete probability distribution, and a decoder that takes the encoded embedding for reconstructing the input.

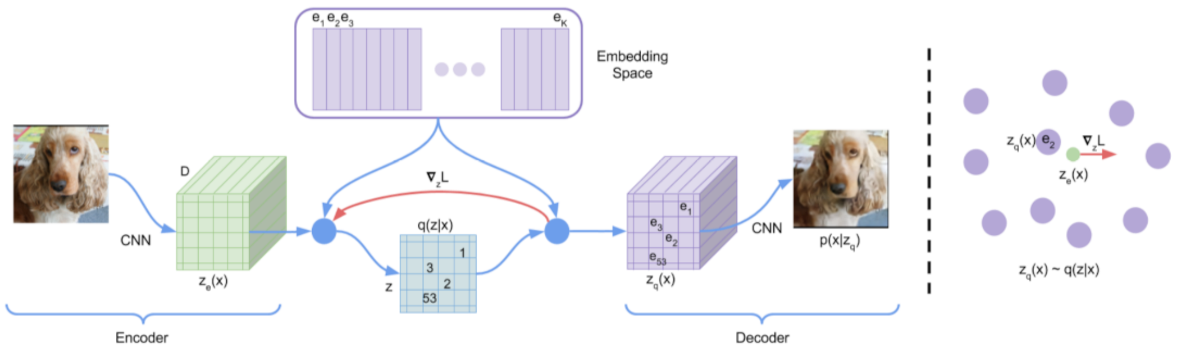


Figure 9.4: Encoder-decoder architecture of VQ-VAE [104]

So, the process for each example is as follows:

1. The encoder obtains a set of vector features from the input example
2. The vector features are matched using the euclidean distance to the nearest code of the dictionary of symbols (each symbol corresponds to a centroid) obtaining a discrete representation
3. The codes are substituted by the centroid vectors of the dictionary and transformed again to samples by the decoder

All this is optimized using gradient descent, so the loss has to include the computation of the dictionary apart from the reconstruction of the samples. The loss in this case is composed of three terms, one is the *reconstruction loss* that is the L_2 between the sample x and the result of the decoding of its quantization $D(e)$, the second is *codebook loss* that only affect to the codebook variables and is the L_2 between the codebook embedding vectors e and the output of the encoder $E(x)$, that makes the vectors from the codebook to stay close to the output of the encoder and the last one is the *commitment loss* that only affects to the encoder weights and is the L_2 between the codebook embedding vectors e and the output of the encoder $E(x)$, this encourages the output of the encoder to be close to the selected vector from the codebook, so it does not fluctuate from one to another (the *sg* operator blocks the gradients flow).

$$\mathcal{L}(x, D(e)) = ||x - D(e)||_2^2 + ||sg(E(x) - e||_2^2 + \beta||e - sg(E(x)||_2^2$$

For inference the model also needs to define a discrete prior distribution over the latent codes to generate new samples. This is implemented with an autoregressive distribution fitted after training the VQ-VAE with new encoded samples using ancestral sampling. The advantage is that we are working on the latent space that has reduced largely the dimensionality of the problem, so it is more efficient for this autoregressive training.

VQ-VAE2 performs some improvements to the original model. In this case they use a hierarchical/multiscale approach by generating two quantized images with two resolutions (Top + bottom level). Each level depends on pixels and the top level is conditioned on the bottom level. Both quantizations are passed to the decoder for generating the image.

The encoder is composed by residual blocks. The decoder is similar, with also residual blocks and strided transposed convolutions. The network is trained with a similar loss as the previous version of the model.

It has also a second training stage that computes a prior distribution over the latent codes in order to sample from the network. For modelling this prior they chose as autoregressive

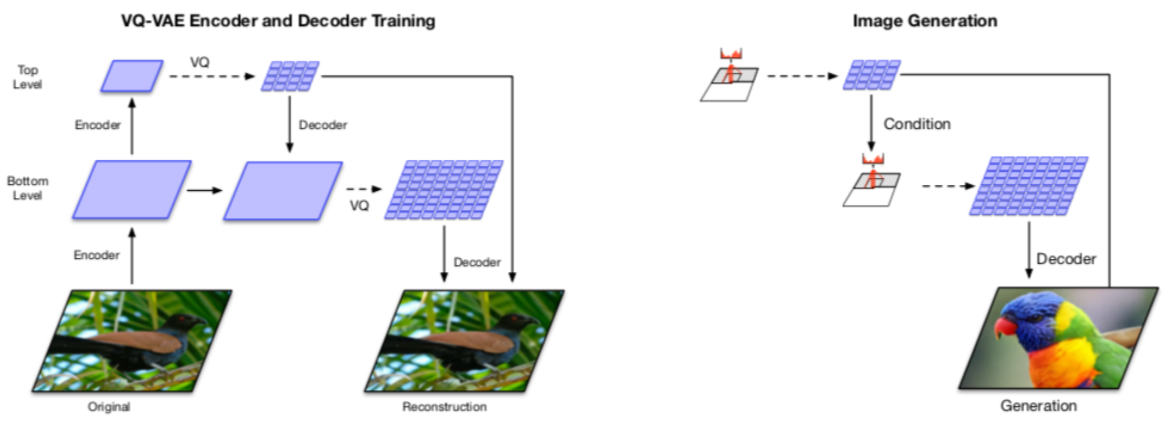


Figure 9.5: Encoder-decoder hierarchical architecture of VQ-VAE2 [111]

model PixelCNN adding different modifications. The residual gated layers of PixelCNN are interleaved with Multi Headed attention and dropout is used after each residual block. Deep residual networks with 1×1 convolutions are added on top. The model can also use the class of the images as part of the prior distribution for conditional generation.

9.7 Notebook: Variational Auto Encoders (VAEs)

In this notebook we will use different types of VAEs. As you will remember, these models assume that there is a latent distribution of fewer dimensions that generates the observed variables. The objective is to estimate this distribution using an auto-encoder architecture, where the encoder estimates the distribution of the latent variables and the decoder is capable of generating samples from these variables.

We will use the `pythae` library which implements many variants of these models. This library makes building and training models quite easy.

First we will redefine a couple of classes for training the models to make it somewhat less verbose than the class that comes by default in the library simply to make the notebook results more readable.

9.7.1 VAE MNIST (MLP Encoder-Decoder)

We will start with a VAE that uses MLPs in the encoder and decoder. We will load the MNIST (handwritten digits) data set, but we will only use a part of the data so as not to make the training of the models too long. Images are 28x28 in size and pixels are in the range [0-255]

We will select the first 5000 examples as the training set and the last 5000 for evaluation (purely arbitrary selection, the data size can be reduced so that training takes less time, but obviously the quality of the results will decrease)

[7]:
```
dsize = 5000
train_dataset = mnist_trainset.data[:dsize].reshape(-1, 1, 28, 28) / 255.
train_labels =  mnist_trainset.targets[:dsize]
eval_dataset = mnist_trainset.data[-dsize:].reshape(-1, 1, 28, 28) / 255.
eval_labels =  mnist_trainset.targets[-dsize:]
digdata =  BaseDataset(train_dataset, train_labels)
```

```
digeval =  BaseDataset(eval_dataset, eval_labels)
```

In this library we must first create an object that allows us to configure some elements of the model, in this case the dimensions that the examples have and the number of latent variables that the VAE will have. Obviously the chosen value seems small, you can experience the effect of increasing the number of latents.

[8]:
```
vae_MLP_config = VAEConfig(
    input_dim=(1, 28, 28),
     latent_dim=8
 )
```

We will create a classical VAE that assumes that the latent variables correspond to Gaussian (prior) distributions. In this case, if we do not pass the networks for the encoder and the decoder, it will use an MLP with default dimensions.

[9]:
```
vae_MLP_model = VAE(
     model_config=vae_MLP_config
 )
```

If we look at the model we can see that the decoder has two layers, one transforms the set of latent variables to a layer of 512 and another gives us the output of 28x28 using ReLUs as the activation function and a sigmoid as the output to give us values in the range [0-1].

The decoder has a 512 layer with a ReLU activation function and two outputs, one obtains the mean of the Gaussian latent variables and the other the logarithm of the variances. From here the model will be able to calculate the ELBO and the NLL.

[10]:
```
vae_MLP_model
```

[10]:
```
VAE(
  (decoder): Decoder_AE_MLP(
    (layers): ModuleList(
      (0): Sequential(
        (0): Linear(in_features=8, out_features=512, bias=True)
        (1): ReLU()
      )
      (1): Sequential(
        (0): Linear(in_features=512, out_features=784, bias=True)
        (1): Sigmoid()
      )
    )
  )
  (encoder): Encoder_VAE_MLP(
    (layers): ModuleList(
      (0): Sequential(
        (0): Linear(in_features=784, out_features=512, bias=True)
        (1): ReLU()
      )
    )
    (embedding): Linear(in_features=512, out_features=8, bias=True)
    (log_var): Linear(in_features=512, out_features=8, bias=True)
  )
)
```

We will train the number of epochs that has been defined in the variable `n_epochs` using AdamW and reducing the learning rate if the loss function stagnates. Notice that we are using a different *scheduler* than in previous notebooks.

[11]:
```
training_config = BaseTrainerConfig(
   output_dir='my_model',
    num_epochs=n_epochs,
    learning_rate=1e-3,
    per_device_train_batch_size=256,
    per_device_eval_batch_size=256,
    steps_saving=None,
    optimizer_cls="AdamW",
    optimizer_params={"weight_decay": 0.05, "betas": (0.91, 0.995)},
    scheduler_cls="ReduceLROnPlateau",
    scheduler_params={"patience": 5, "factor": 0.9}
)
```

We define the training object

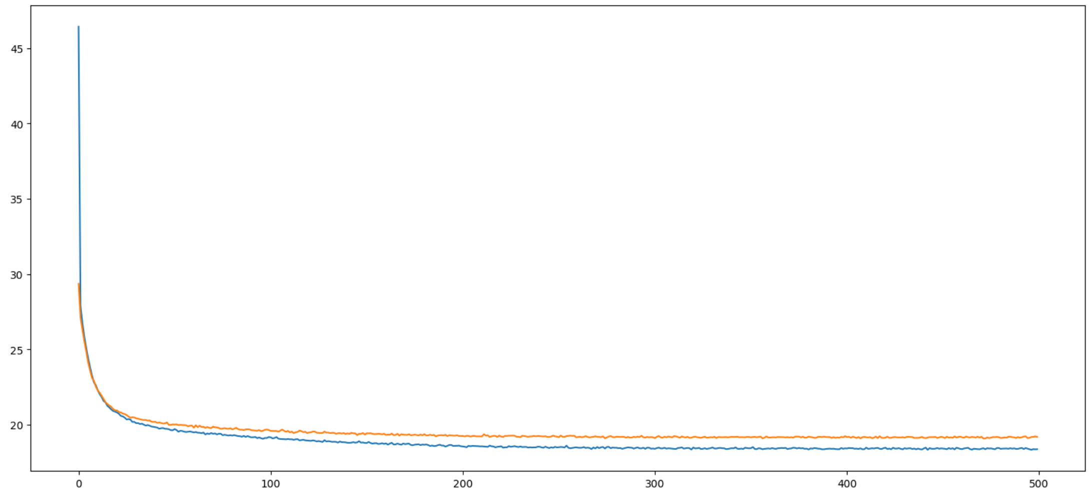
[12]:
```
mcp = TrainHistoryCallback()
trainer = BaseTrainerS(model=vae_MLP_model,
                      training_config=training_config,
                      train_dataset=digdata,
                      eval_dataset= digeval,
                      callbacks=[mcp, ProgressBarEpochCallback()])
```

and fit the model

[13]:
```
trainer.train()
```

[14]:
```
plt.figure(figsize=(18,8))
print(mcp.history['train_loss'][-1])
plt.plot(mcp.history['train_loss']);
plt.plot(mcp.history['eval_loss']);
```

```
18.367655658721922
```



We can calculate the log likelihood of the model for a sample, this allows us to compare the models

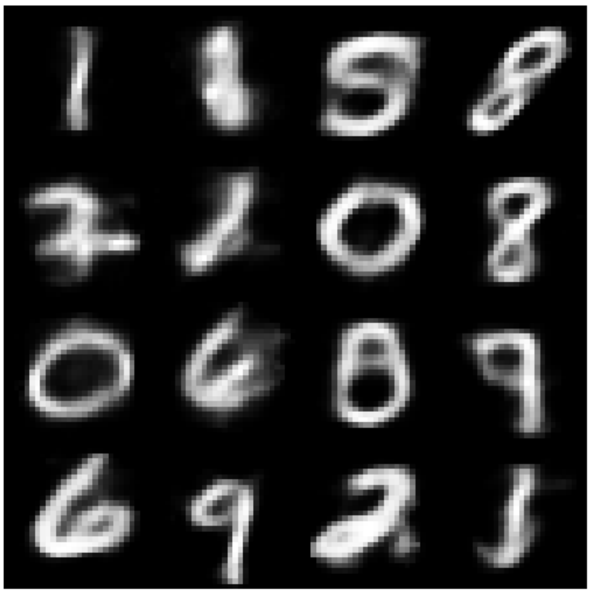
[15]:
```
vae_MLP_model.get_nll(train_dataset.to(device))
```

[15]:
```
-738.4649844017599
```

Now we can generate samples. In this case we will obtain values by sampling the Gaussian distributions that correspond to the parameters that the VAE has learned and we will pass them through the decoder. If the data has fit enough to the Gaussian distribution that we use as a priori distribution we should have something that makes sense.

[16]:
```
gpipeline = GenerationPipeline(vae_MLP_model)
```

[17]:
```
nim = 4
samples = gpipeline(nim*nim)
fig = plt.figure(figsize=(5,5))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow((samples[i].cpu().detach().permute(1,2,0).numpy()),␣
 ↪cmap='gray')
    plt.axis('off')
plt.tight_layout(pad=0.)
```



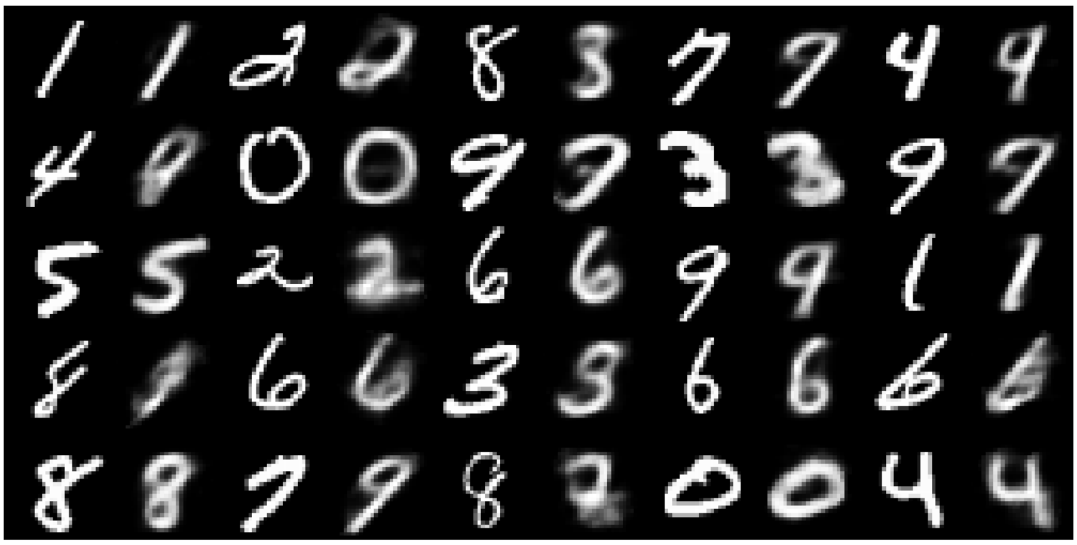
We can see that the digits are slightly blurry. This is a common problem in VAEs trained with the Gaussian distribution as a priori distribution.

We can also check if the VAE is capable of reconstructing examples (we have trained it for that too) using the validation data.

[18]:
```
reconstructions = vae_MLP_model.reconstruct(eval_dataset[:25].to(device)).
 ↪detach().cpu()
```

[19]:
```
fig, axes = plt.subplots(nrows=5, ncols=10, figsize=(10, 5))

for i in range(5):
    for j in range(5):
        axes[i][2*j].imshow(eval_dataset[i*5 + j].cpu().squeeze(0),␣
 ↪cmap='gray')
```

```
        axes[i][2*j].axis('off')
        axes[i][2*j+1].imshow(reconstructions[i*5 + j].cpu().squeeze(0),␣
↪cmap='gray')
        axes[i][2*j+1].axis('off')

plt.tight_layout(pad=0.)
```



The results are not fantastic, but in most cases the reconstruction at least matches the digit.

9.7.2 VAE MNIST (ResNet Encoder-Decoder)

Let's now change the network architecture to use a residual network (ResNet) instead of an MLP. The library already has a predefined network for each part of the network. The encoder reduces the dimensions of the input with convolutions (downscaling) until reaching a layer that is transformed into the mean and log variance of the latent variables and a decoder that transforms the latent variables by rescaling (upscaling) with convolutions transposed in the image. final

[20]:
```
from pythae.models.nn.benchmarks.mnist import Encoder_ResNet_VAE_MNIST,␣
↪Decoder_ResNet_AE_MNIST
```

[21]:
```
vae_Resnet_config = VAEConfig(
    input_dim=(1, 28, 28),
     latent_dim=8
)
```

We define the model the same as before, but now we pass it the new networks for the encoder and the decoder.

[22]:
```
vae_Resnet_model = VAE(
     model_config=vae_Resnet_config,
    encoder=Encoder_ResNet_VAE_MNIST(vae_Resnet_config),
    decoder=Decoder_ResNet_AE_MNIST(vae_Resnet_config)
)
```

We train the same way

[25]:
```
training_config = BaseTrainerConfig(
   output_dir='my_model',
    num_epochs=n_epochs,
    learning_rate=1e-3,
    per_device_train_batch_size=256,
    per_device_eval_batch_size=256,
    steps_saving=None,
    optimizer_cls="AdamW",
    optimizer_params={"weight_decay": 0.05, "betas": (0.91, 0.995)},
    scheduler_cls="ReduceLROnPlateau",
    scheduler_params={"patience": 5, "factor": 0.9}
 )
```

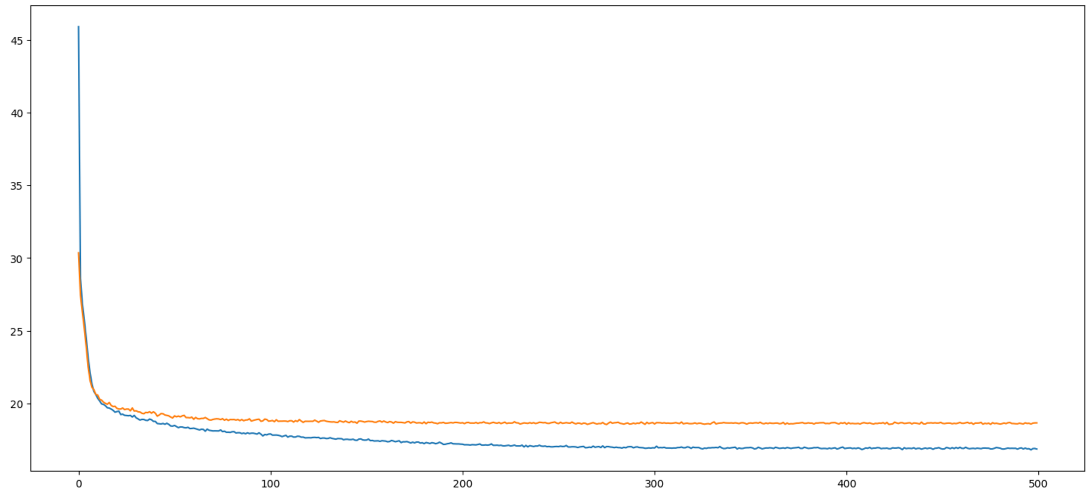
[26]:
```
mcp = TrainHistoryCallback()
trainer = BaseTrainerS(model=vae_Resnet_model,
                      training_config=training_config,
                      train_dataset=digdata,
                      eval_dataset=digeval,
                      callbacks=[mcp, ProgressBarEpochCallback()])
```

[27]:
```
trainer.train()
```

We obtain a relative gain in the loss function

[28]:
```
plt.figure(figsize=(18,8))
print(mcp.history['train_loss'][-1])
plt.plot(mcp.history['train_loss']);
plt.plot(mcp.history['eval_loss']);
```

```
16.862343978881835
```



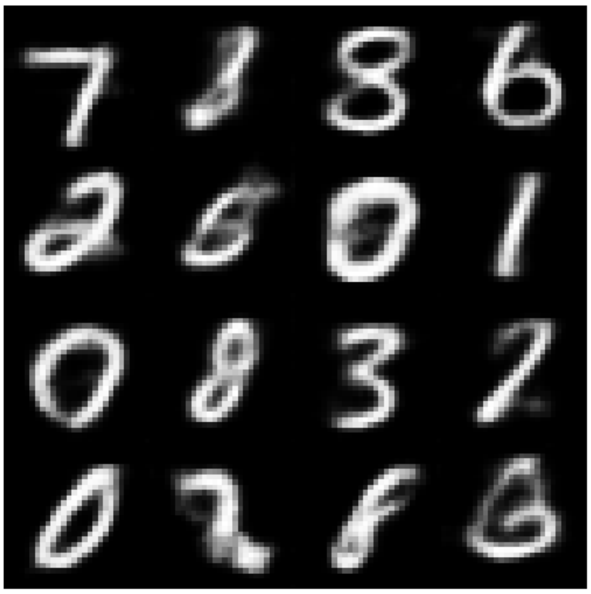
The NLL is not very different than in the previous model

[29]:
```
vae_Resnet_model.get_nll(train_dataset.to(device))
```

[29]:
```
-736.7897817336673
```

[30]:
```
gpipeline = GenerationPipeline(vae_Resnet_model)
```

[31]:
```
nim = 4
samples = gpipeline(nim*nim)
fig = plt.figure(figsize=(5,5))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(samples[i].cpu().detach().permute(1,2,0).numpy(), cmap='gray')
    plt.axis('off')
plt.tight_layout(pad=0.)
```

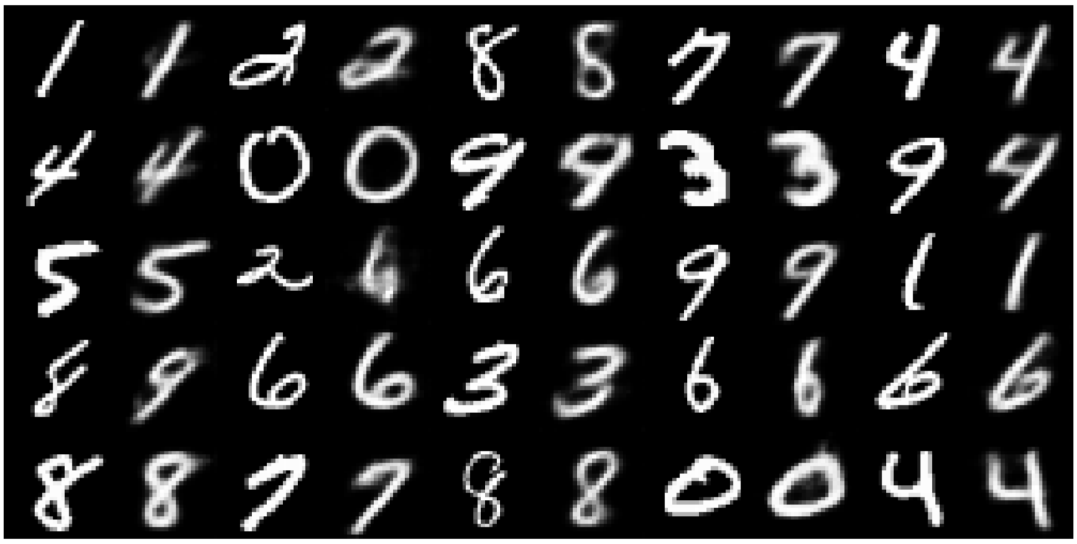


The samples are a little better, probably a little less blurry than what we got with the previous VAE.

We can see how it does this by rebuilding the validation set.

[32]:
```
reconstructions = vae_Resnet_model.reconstruct(eval_dataset[:25].to(device)).
 ↪detach().cpu()
```

[33]:
```
fig, axes = plt.subplots(nrows=5, ncols=10, figsize=(10, 5))

for i in range(5):
    for j in range(5):
        axes[i][2*j].imshow(eval_dataset[i*5 + j].cpu().squeeze(0),␣
 ↪cmap='gray')
        axes[i][2*j].axis('off')
        axes[i][2*j+1].imshow(reconstructions[i*5 + j].cpu().squeeze(0),␣
 ↪cmap='gray')
        axes[i][2*j+1].axis('off')

plt.tight_layout(pad=0.)
```



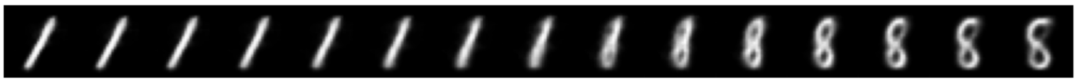
The results seem better than the first VAE

We can also take advantage of this to test a quality that VAEs also have, which is the possibility of interpolating between examples. In this case the interpolation space is the space defined by the latent variables, which is of less dimensionality than in the flows.

We are going to interpolate between two real samples. To do this, the method is to encode the source and destination sample and obtain a vector that connects these interpolations in the latent space. We sample the line that connects them at regular intervals. These points are passed through the decoder to obtain the interpolated samples.

[34]:
```
ninter = 15
interpolations = vae_Resnet_model.interpolate(eval_dataset[0].to(device),
                                              eval_dataset[2].to(device).
 ↪unsqueeze(0),
                                              granularity=ninter).squeeze(0).
 ↪detach().cpu()
```

[35]:
```
fig, axes = plt.subplots(nrows=1, ncols=ninter, figsize=(10, 5))
axes

for i in range(ninter):
        axes[i].imshow(interpolations[i], cmap='gray')
        axes[i].axis('off')

plt.tight_layout(pad=0.)
```



If the VAE has learned a good transformation the transition should be relatively smooth.

9.7.3 VQ VAE

The Vector Quantized VAE (not really a VAE) makes a discretization of the space by assigning to the input a vector from a limited set of vectors that is adjusted during training to obtain the best reconstruction.

[36]:
```
from pythae.models import VQVAE, VQVAEConfig
```

[37]:
```
vqvae_config = VQVAEConfig(
    input_dim=(1, 28, 28),
    latent_dim=8,
    commitment_loss_factor=0.25,
    quantization_loss_factor=1.0,
    num_embeddings=512,
    use_ema=True,
    decay=0.99
 )
```

[38]:
```
vqvae_model = VQVAE(
     model_config=vqvae_config,
 )
```

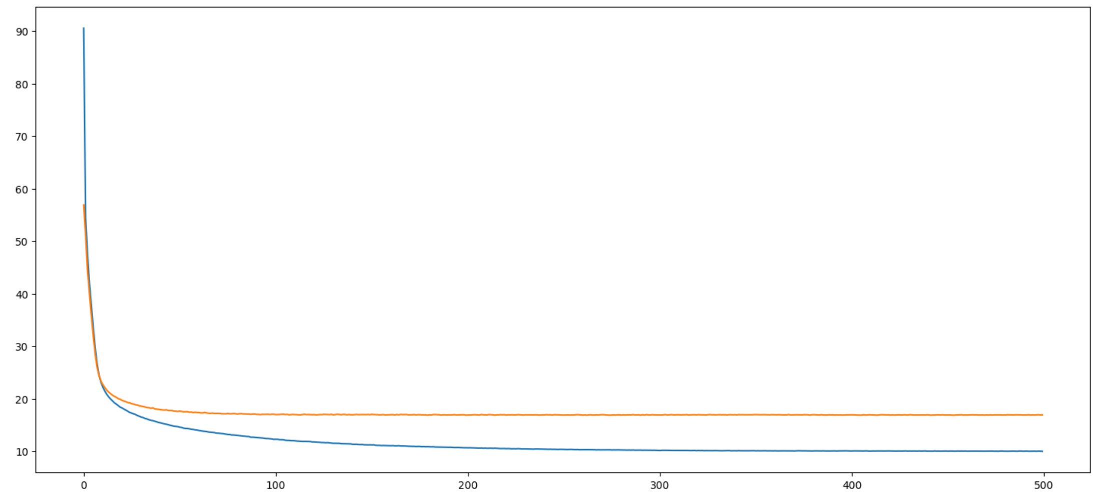
[39]:
```
training_config = BaseTrainerConfig(
   output_dir='my_model',
    num_epochs=n_epochs,
    learning_rate=1e-3,
    per_device_train_batch_size=256,
    per_device_eval_batch_size=256,
    steps_saving=None,
    optimizer_cls="AdamW",
    optimizer_params={"weight_decay": 0.05, "betas": (0.91, 0.995)},
    scheduler_cls="ReduceLROnPlateau",
    scheduler_params={"patience": 5, "factor": 0.9}
 )
```

[40]:
```
mcp = TrainHistoryCallback()
trainer = BaseTrainerS(model=vqvae_model,
                      training_config=training_config,
                      train_dataset=digdata,
                      eval_dataset=digeval,
                      callbacks=[mcp, ProgressBarEpochCallback()])
```

[41]:
```
trainer.train()
```

[42]:
```
plt.figure(figsize=(18,8))
print(mcp.history['train_loss'][-1])
plt.plot(mcp.history['train_loss']);
plt.plot(mcp.history['eval_loss']);
```

```
9.995860719680786
```



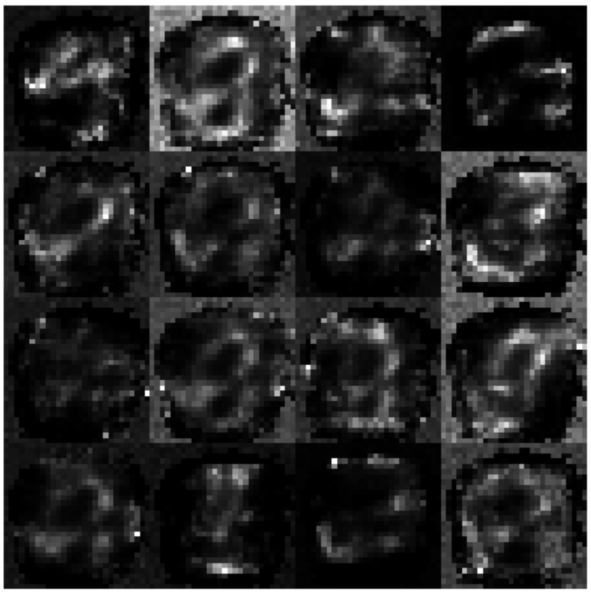
We can see that the loss function is somewhat better.

In this case we cannot calculate the NLL since VQVAE is not really a VAE

If we sample

[43]:
```
gpipeline = GenerationPipeline(vqvae_model)
```

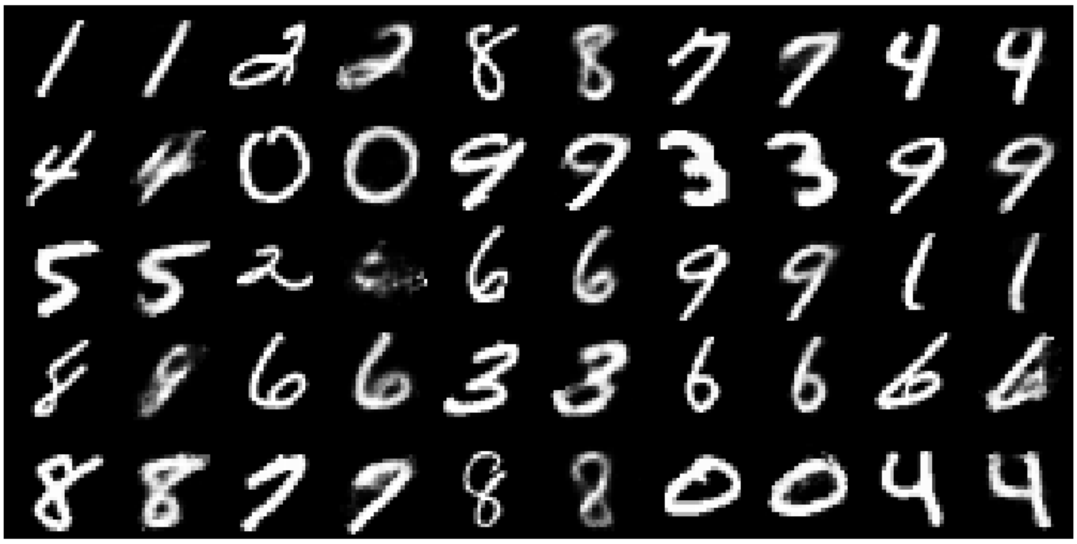
[44]:
```
nim = 4
samples = gpipeline(nim*nim)
fig = plt.figure(figsize=(5,5))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(samples[i].cpu().detach().permute(1,2,0).numpy(), cmap='gray')
    plt.axis('off')
plt.tight_layout(pad=0.)
```



The sampling is frankly terrible, this is basically because we are using a Gaussian distribution when the set of vectors that quantize the space does not follow this distribution.

[45]:
```
reconstructions = vqvae_model.reconstruct(eval_dataset[:25].to(device)).
  ↪detach().cpu()
```

[46]:
```
fig, axes = plt.subplots(nrows=5, ncols=10, figsize=(10, 5))

for i in range(5):
    for j in range(5):
        axes[i][2*j].imshow(eval_dataset[i*5 + j].cpu().squeeze(0),␣
  ↪cmap='gray')
        axes[i][2*j].axis('off')
        axes[i][2*j+1].imshow(reconstructions[i*5 + j].cpu().squeeze(0),␣
  ↪cmap='gray')
        axes[i][2*j+1].axis('off')

plt.tight_layout(pad=0.)
```



On the other hand, we have better reconstructions (with exceptions, which are also difficult for the other VAEs) and they are no longer blurred due to the quantization.

9.7.4 Sampling with Gausian Mixture Models

As we commented, VAEs can have a mismatch problem between the a priori distribution and the distribution generated by the decoder so that probability masses that should be probable are not probable in the generation and probability masses that have little probability in data is generated more frequently.

We can correct this by training a model to do the sampling that readjusts the distribution based on a more complex a priori distribution.

One way to improve sampling capacity is to train a model that better fits the distribution followed by quantization, so that the samples do not come from a Gaussian (the one we use by default) but from a more complex distribution.

For example, we can consider that the distribution we have corresponds to a mixture of Gaussians and estimate its parameters, so that the different modes of the Gaussians are concentrated in the parts of the space where the examples that the decoder is capable of generating are located. Now samples outside of those modes will be less likely to be generated.

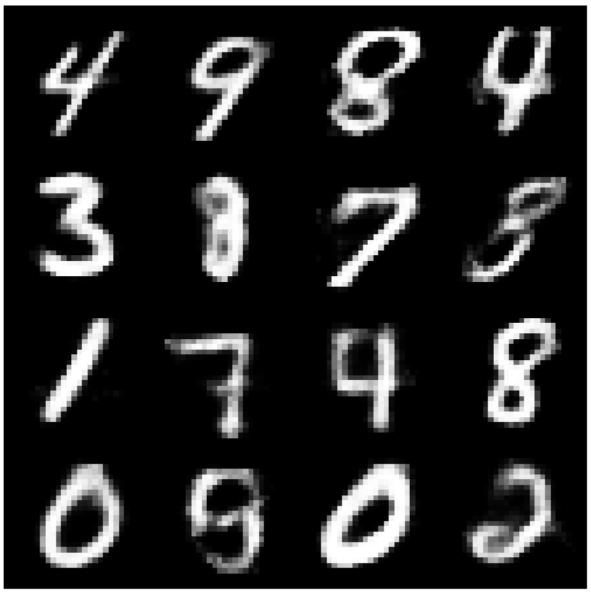
20 has been chosen as the size of the Gaussian mixture arbitrarily, but you can test the effect of using more or fewer components. The sampler will adjust a GMM to the latents of the training data and sampling will be done by obtaining samples according to the learned distribution.

[47]:
```
from pythae.samplers import GaussianMixtureSampler,␣
↪GaussianMixtureSamplerConfig, IAFSamplerConfig, IAFSampler,␣
↪PixelCNNSampler, PixelCNNSamplerConfig
```

[48]:
```
# set up GMM sampler config
gmm_sampler_config = GaussianMixtureSamplerConfig(
    n_components=20
)

# create gmm sampler
gmm_sampler = GaussianMixtureSampler(
    sampler_config=gmm_sampler_config,
    model=vqvae_model
)

# fit the sampler
gmm_sampler.fit(train_dataset)
```

If we sample the distribution

[49]:
```
nim = 4
samples =  gmm_sampler.sample(num_samples=nim*nim)
fig = plt.figure(figsize=(5,5))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(samples[i].cpu().detach().permute(1,2,0).numpy(), cmap='gray')
    plt.axis('off')
plt.tight_layout(pad=0.)
```



Now we do obtain samples of minimally recognizable digits.

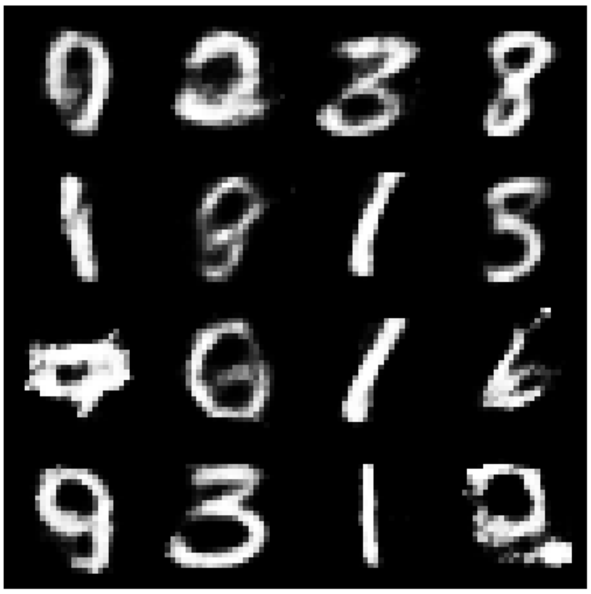
9.7.5 VAE with normalising flow

The GMM does not give enough freedom when adjusting an a priori distribution to allow better sampling. One possibility is to learn a normalizing flow that does the transformation between the Gaussian prior distribution and the one needed by the decoder.

Let's try using a reverse Auto Regressive Flow. This has the advantage over the autoregressive flow that it is faster generating samples since we learn it in reverse than usual (from a priori distribution to the data).

[50]:
```
training_config = BaseTrainerConfig(
   output_dir='my_model',
    num_epochs=100,
    learning_rate=1e-3,
    per_device_train_batch_size=256,
    per_device_eval_batch_size=256,
    steps_saving=None,
    optimizer_cls="AdamW",
    optimizer_params={"weight_decay": 0.05, "betas": (0.91, 0.995)},
    scheduler_cls="ReduceLROnPlateau",
    scheduler_params={"patience": 5, "factor": 0.9}
)

# set up IAF sampler config
IAF_sampler_config = IAFSamplerConfig(
    n_made_blocks =2,
    n_hidden_in_made = 3,
    hidden_size = 1024
)

# create gmm sampler
IAF_sampler = IAFSampler(
    sampler_config=IAF_sampler_config,
    model=vqvae_model
)

# fit the sampler
IAF_sampler.fit(train_dataset,eval_data=eval_dataset,␣
 ↪training_config=training_config)
```

[51]:
```
nim = 4
samples =  IAF_sampler.sample(num_samples=nim*nim)
fig = plt.figure(figsize=(5,5))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(samples[i].cpu().detach().permute(1,2,0).numpy(), cmap='gray')
    plt.axis('off')
plt.tight_layout(pad=0.)
```



9.7.6 β-VAE

With VAEs you can also try to force the latent variables to correspond to variables with some meaning in the domain. The simplest of these methods is β-VAE which adds a weight to the VAE loss function so that we have a weight that changes the influence of the regularization. Depending on how we adjust that weight, we can obtain a VAE where the first variables control elements of the samples.

[52]:
```
from pythae.models import BetaVAEConfig
from pythae.models import BetaVAE
```

We can change the β parameter of the VAE to force further "untangling" of the latent variables. In this VAE it is difficult to control, but we can see if we obtain a VAE where by exploring the range of the first latent variable and fixing the rest something meaningful changes in the sample.

Depending on the parameter value we can make the VAE perform worse, so it needs to be adjusted carefully. (a range between 1 and 2 should give consistent results)

Using a value of 1 we obviously have the original VAE.

[53]:
```
bvae_Resnet_config = BetaVAEConfig(
    input_dim=(1, 28, 28),
     latent_dim=8,
     beta=1.5
 )
```

[54]:
```
bvae_Resnet_model = BetaVAE(
     model_config=bvae_Resnet_config,
    encoder=Encoder_ResNet_VAE_MNIST(bvae_Resnet_config),
    decoder=Decoder_ResNet_AE_MNIST(bvae_Resnet_config)
 )
```

[55]:
```
training_config = BaseTrainerConfig(
   output_dir='my_model',
    num_epochs=n_epochs,
    learning_rate=1e-3,
    per_device_train_batch_size=256,
    per_device_eval_batch_size=256,
    steps_saving=None,
    optimizer_cls="AdamW",
    optimizer_params={"weight_decay": 0.05, "betas": (0.91, 0.995)},
    scheduler_cls="ReduceLROnPlateau",
    scheduler_params={"patience": 5, "factor": 0.9}
)
```

We define the object for training

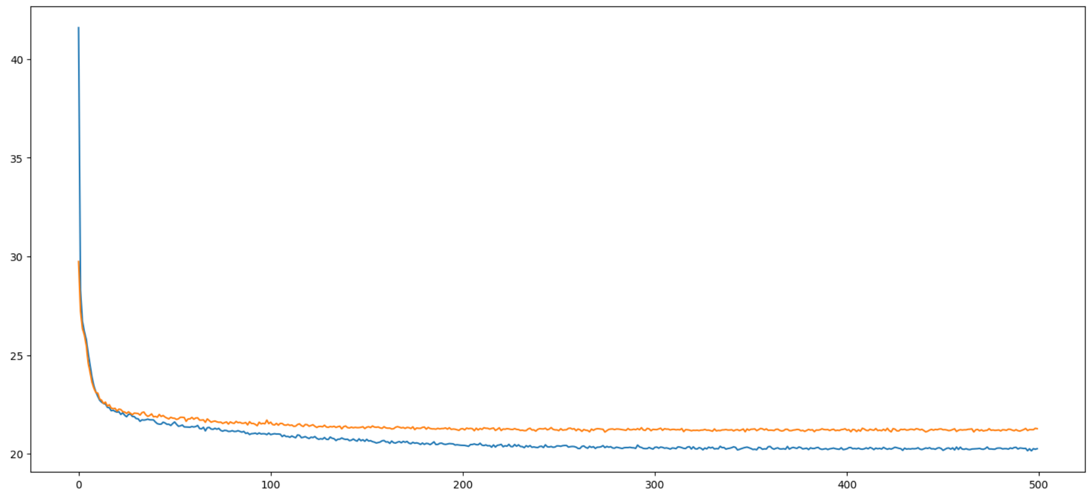
```
[56]: mcp = TrainHistoryCallback()
      trainer = BaseTrainerS(model=bvae_Resnet_model,
                             training_config=training_config,
                             train_dataset=digdata,
                             eval_dataset= digeval,
                             callbacks=[mcp, ProgressBarEpochCallback()])
```

and we train the model

```
[57]: trainer.train()
```

```
[58]: plt.figure(figsize=(18,8))
      print(mcp.history['train_loss'][-1])
      plt.plot(mcp.history['train_loss']);
      plt.plot(mcp.history['eval_loss']);
```

```
20.259890747070312
```



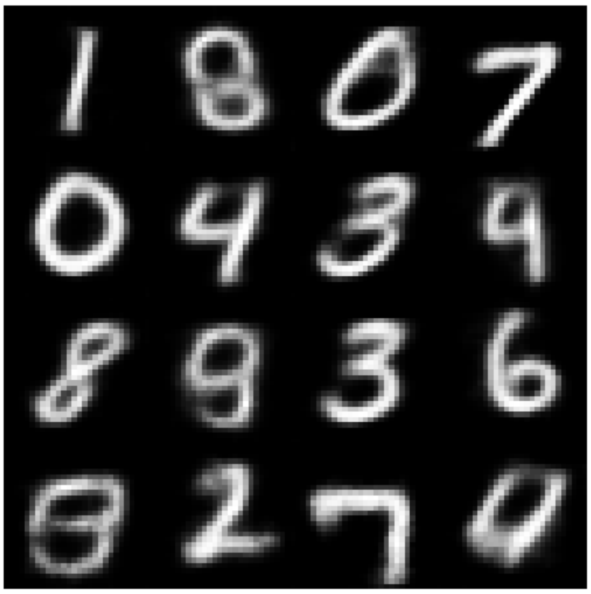
```
[59]: bvae_Resnet_model.get_nll(train_dataset.to(device))
```

```
[59]: -737.3598185738132
```

If we haven't gone too far, the samples should be minimally decent.

```
[60]: gpipeline = GenerationPipeline(bvae_Resnet_model)
      nim = 4
      samples = gpipeline(nim*nim)
```

```
fig = plt.figure(figsize=(5,5))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow((samples[i].cpu().detach().permute(1,2,0).numpy()),␣
 ↪cmap='gray')
    plt.axis('off')
plt.tight_layout(pad=0.)
```



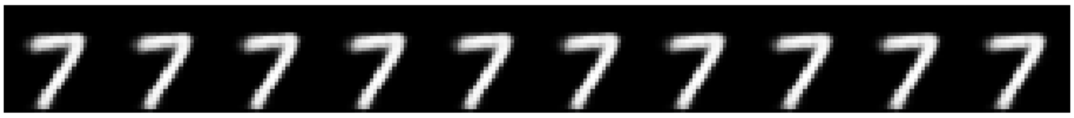
We will need the VAE decoder to be able to experiment with the latents

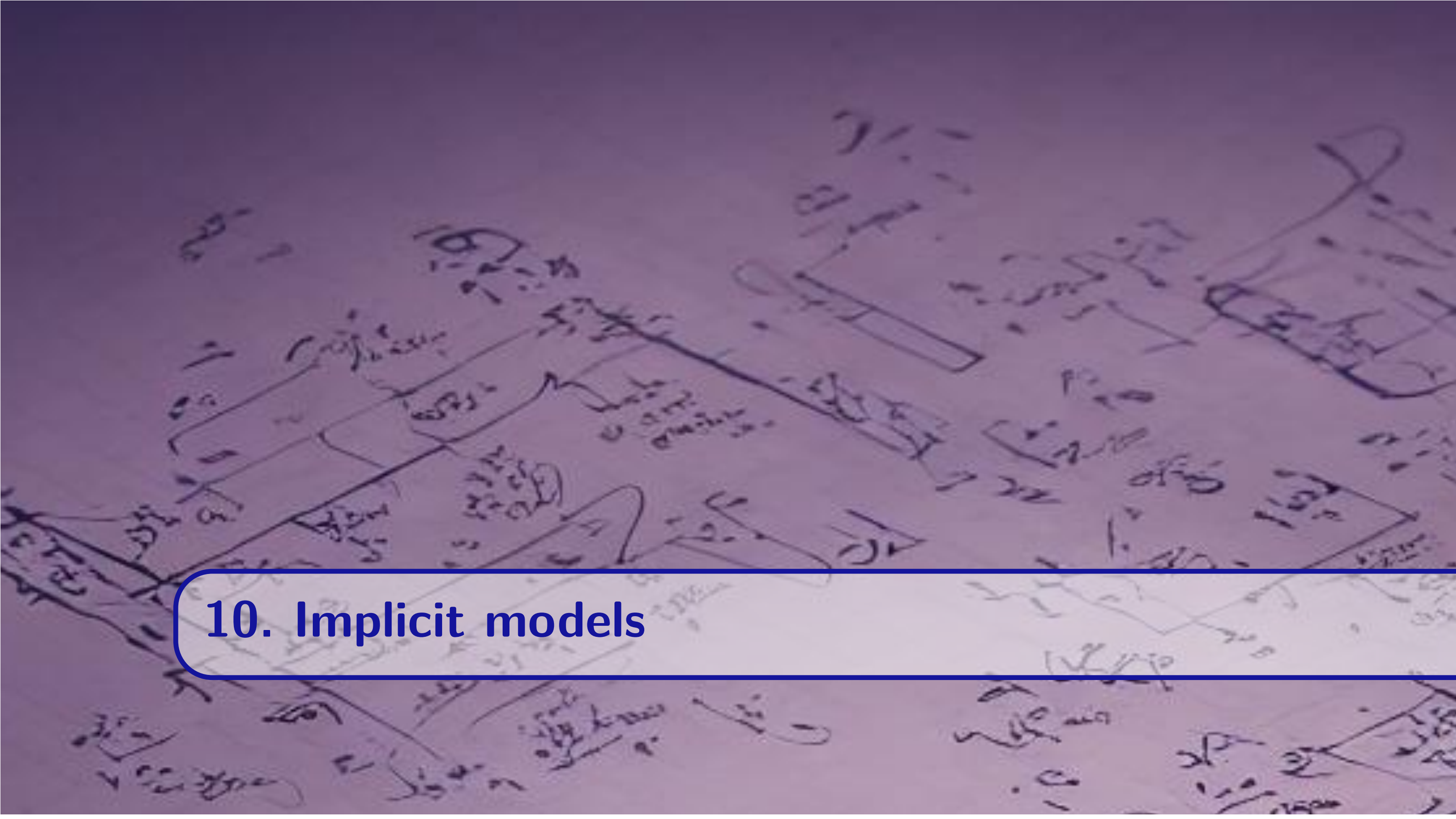
```
[61]: bvaed = bvae_Resnet_model.decoder
```

We generate a Gaussian vector and change the value of the first position in a range of values at regular intervals in the hope that it controls, for example, the digit that is obtained or its shape. It is also possible that what we obtain will not be consistent, and would require a more exhaustive hyperparameter exploration.

Obviously, depending on the noise vector that is generated, different things will come out each time you run the cell, you can replace it with a vector of zeros to make it reproducible.

```
[62]: noise = torch.randn((1,8))
fig = plt.figure(figsize=(10,15))
for i, v in enumerate(torch.arange(-2,2,0.4)):
    noise[0,0] = v
    samp = bvaed(noise.to(device))['reconstruction'].cpu().detach().squeeze()
    plt.subplot(1, 11, i+1)
    plt.imshow(samp, cmap='gray')
    plt.axis('off')
plt.tight_layout(pad=0.)
```





10. Implicit models

10.1 Introduction

All the models that we have talked about in the previous chapters are based on fitting a probability distribution using likelihood. We can obtain exact likelihoods as in autoregressive models and flows or approximate ones using a variational distribution and optimizing the ELBO. Once trained we can obtain new samples transforming samples from a prior distribution.

For some problems we could only be interested on obtaining samples, without the additional information that comes from fitting a probability distribution, like, for instance, to do data augmentation. Sometimes we do not need an explicit model for the data. This opens the possibility of having models only for sampling, without a connection with the likelihood of a distribution.

Having this is mind, we could define what we want from a model that only generates samples. For starters, we do not want a model that just generates variations of the training data. We also want to be able to interpolate between the samples in the training data and to be able to obtain intermediate samples smoothly. We could also ask for model that is aware of the factors that generate the data, basically that the learning performs some kind of disentanglement as we saw it is possible for VAEs. This would allow using the model to modify or obtain variations that are only affected by these factors, like changes in viewing angles, illumination, color, shape...

In order to introduce the capability for sampling in these models, we need to introduce a reference probability distribution z. This distribution does not need to have the same dimensionality as the samples, and we can choose any simple distribution as for instance gaussian or uniform. The model will transform the samples from the reference distribution to samples that will resemble samples of the distribution of the training data. This transformation will be performed by a neural network.

Formally, given a training dataset that represents the data distribution, we will obtain samples from a reference distribution $z \sim p(z)$ that will be processed by a neural network for obtaining a distribution $q_\phi(z) = DNN(z;\phi)$. The neural network will represent implicitly a distribution that will correspond to the distribution of the problem.

The goal will be to obtain a network that learns the distribution from the training data so the distribution that we observe on the data is as close as possible to the distribution that is generated by the sampling of the network.

The hard problem is that, differently from the previous methods, we do not have an expression for the learned distribution, so we can not use likelihood for optimization or use divergence measures (like Kullback-Leibler) to compare them. This means to have to resort to other ways of comparing what is generated and the training samples that do not have a general framework to support our methodology.

10.2 Generative Adversarial Networks

Generative Adversarial Networks (GANs) were defined in 2014 by the paper of the same name [52]. The basic idea is to use two models, one called the **Generator** (G) that captures the distribution of the data and is able to compute samples from a latent vector obtained from a simple distribution and other called the **Discriminator** (D) that computes the probability for a sample of being from the original data or from the generator. The training is performed as an adversarial minimax game with the networks as players. The generator plays for maximizing the probability of its own samples and the discriminator plays for minimizing the probabilities of the generator samples. The discriminator acts as a classifier for a binary problem, so we can assign labels to the real (1) and generated examples (0) and use binary cross entropy to obtain a loss for optimization.

The optimization problem of the game can be defined as:

$$\min_G \max_D \mathbb{E}_{x \sim p_{data}}[\log D(x)] + \mathbb{E}_{z \sim p(z)}[\log(1 - D(G(z)))]$$

Where $D(x)$ is the probability assigned by the discriminator to real samples and $D(G(z))$ the probability assigned to samples obtained from sampling the generator. The discriminator should be optimized for increasing $\mathbb{E}_{x \sim p_{data}}[\log D(x)]$ and the generator for increasing $\mathbb{E}_{z \sim p(z)}[\log(1 - D(G(z)))$. If an equilibrium is reached in the game, the discriminator should not be able to distinguish between real and generated samples.

The algorithm in the original paper defined the training as follows:

```
for a number of iterations do
    for k steps do
        Sample $m$ noise $z$ vectors from the reference distribution
        Sample $m$ examples $x$ from the training data
        Update the discriminator with gradient descent with:
            $\nabla\theta_D \frac{1}{m} \sum_{i=1}^{m} [\log D(x_i) + \log(1 - D(G(z_i)))]$
    end
    Sample $m$ noise $z$ vectors from the reference distribution
    Update the generator with gradient descent with
        $\nabla\theta_G \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z_i)))$
end
```

As you can see, the discriminator can be updated several times each iteration. The basic reason is that at the beginning the discriminator is not good enough for providing information uuld

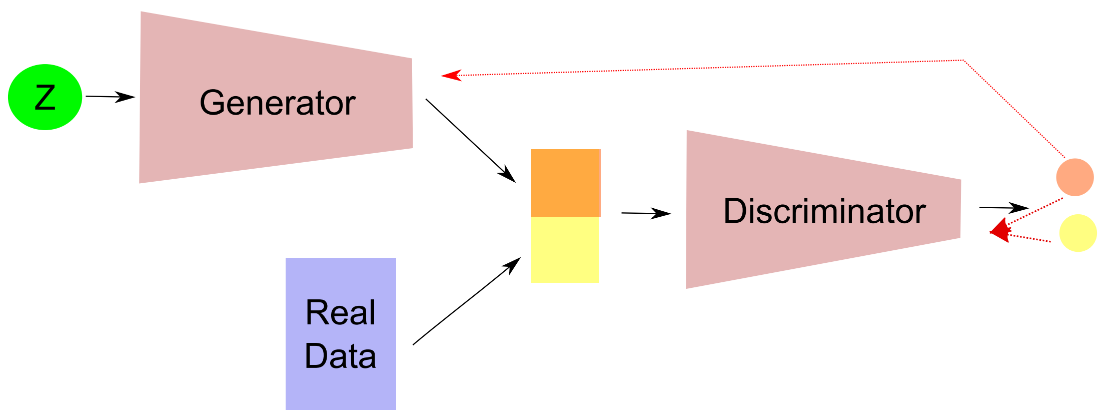


Figure 10.1: Schema of the training of Generative Adversarial Networks

the generator can work with, and it is better to allow the discriminator to improve more quickly to obtain adequate results.

10.2.1 Evaluation of GANs

One hard thing in generative models is to evaluate if the samples that are being generated are faithful to the distribution of the data. In the previous methods we resorted to the likelihood, the better, the closer were the samples to the data distribution. In this case, we can not measure likelihood, and the ability to fool the discriminator is not a guarantee of a good result. It is just a sufficient condition. Think that a generator that only obtains a realistic sample will be good enough for the discriminator. This is actually a problem with these models that is called **mode collapse**. Sometimes, the generator is only able to generate a subset of the real distribution and this can not be detected by the loss.

This means that we need to define measures able to determine how diverse are the samples that are obtained and how well they cover the actual distribution of the data. This is a hard problem given that we are working with data that has very high dimensionality. We would need an exponential number of samples and real data to be able to determine the coverage of the distribution. We could resort to human evaluation, but that is not an objective measure that we can trust.

The first measure that has been used to measure the quality of generated samples on images has been the Inception Score [118]. The idea is to use a reference object classifier for measuring if the content of the generated images is recognizable and if it is diverse. For this we can use a network trained with a large dataset. For this measure the network used is an InceptionV3 that is trained as a classifier of the Imagenet dataset, that contains 1000 classes. This network returns for an image the probability distribution of the labels of the dataset. The measure assumes that if the images are good enough then the network will have high probability of recognizing them as one of the classes of the dataset. If it is not the case, then the probability will be distributed uniformly through all the classes. This mean that if the images are not very diverse the distribution of the prediction will have low entropy (uniform distribution).

The inception score is computed using a large sample of images (>30,000) from the generator to reduce variability. The samples will be classified by the network to obtain the class distribution $p(y|x)$. We can approximate the marginal distribution of the classes $p(y)$ by averaging the distribution of all the classes as:

$$\hat{p}(y) = \frac{1}{N}\sum_i p(y|x_i)$$

From this, the Inception Score is computed using the KL divergence between these two distributions:

$$IS(x) = \exp(\frac{1}{N}\sum_i KL(p(y|x_i)||\hat{p}(y))) = \exp(H(y) - H(y|x))$$

This is also not enough for computing the diversity of the samples from the generator, since it is enough to generate one sample for each mode to obtain a good result. To complement this measure it was defined the **Fréchet Inception Distance** [62]. This measure also uses a pretrained InceptionV3 network, but as a feature extractor. Images from the real and generated data are passed through this network and the activations from the third pooling layer are used as features (2048). The idea is that the distribution of these features for the two sets of examples should be close for their distribution to be similar. This similarity is computed from the mean and the covariance from the extracted features, assuming that they follow a gaussian distribution. This is computed as:

$$d^2((\mu,\Sigma),(\mu_w,\Sigma_w)) = ||\mu - \mu_w||_2^2 + tr(\Sigma + \Sigma_w - 2(\Sigma\Sigma_w)^{\frac{1}{2}})$$

This can be interpreted as a kernelized distance between distributions, where the network is used for computing the kernel.

These measures are obviously not enough guarantee for diversity and coverage, so other measures have been developed. They follow a similar philosophy, by using a pretrained network as feature extractor to be able to operate on a lower dimensionality space. The idea is to approximate the manifold of the real and generated samples in that space. They have the common denominator of using measures based on k-nearest neighbours for approximating the densities in the feature space and measuring something similar to the classical measures of precision and recall from classification.

For instance, precision and recall measures approximating the density using hyperspheres are defined in [116] and [83]. These are computed using k-means clustering or composition of balls defined by k-nearest neighbors. Precision and recall can be computed measuring how each approximation covers the samples from the real and generated data. These measures are sensitive to outliers and computationally costly. The measures of density and coverage are defined in [98] from the approximation of the density only from the real data using counts of generated data that is inside k-neighbor balls and the number of k-neighbor balls that contain generated samples.

10.2.2 Mode collapse

As we have mentioned on the previous section, one of the problems of these models is that they can ignore a part of the probability distribution. This means that some modes (peaks in the distribution) will not be covered by the network implicit distribution. This effect can be theoretically justified by analyzing the behavior of the optimal discriminator, the one that we can obtain if we fully train this network with a fixed generator network.

The Bayes optimal classifier obtained by the original GAN loss optimizes the Jensen-Shannon divergence among the distribution of the generator and the distribution of the real data.

We can show how it is the case by working with the loss of the discriminator, considering the generator

$$\begin{aligned}
V(G,D) &= \mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1-D(G(z)))] \\
&= \int_x p_{data}(x)\log D(x)dx + \int_z p(z)\log(1-D(G(z)))dz \\
&= \int_x p_{data}(x)\log D(x)dx + \int_x p_g(x)\log(1-D(x))dx \\
&= \int_x [p_{data}(x)\log D(x)dx + p_g(x)\log(1-D(x))]dx
\end{aligned}$$

To obtain the optimal bayes classifier we have to obtain the weights that minimizes the loss, thus the weights that make the gradient 0:

$$\nabla_y[a\log y + b\log(1-y)] = 0 \Rightarrow y^* = \frac{a}{a+b}$$

The optimal weights correspond to the probabilities that the data distribution and the generator assign to each sample.

$$D^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p_g(x)}$$

If we substitute this in the loss, and operate

$$\begin{aligned}
V(G,D^*) &= \mathbb{E}_{x\sim p_{data}}[\log D^*(x)] + \mathbb{E}_{z\sim p(z)}[\log(1-D^*(G(z)))] \\
&= \mathbb{E}_{x\sim p_{data}}[\log\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}] + \mathbb{E}_{z\sim p(z)}[\log\frac{p_{data}(x)}{p_{data}(x)+p_g(x)})] \\
&= -log(4) + KL(p_{data}||\frac{p_{data}+p_g}{2}) + KL(p_g||\frac{p_{data}+p_g}{2}) \\
&= -log(4) + JS(p_{data}||p_g)
\end{aligned}$$

we end with the JS divergence plus a constant term $V(G,D^*) = -log(4)$ when $p_{data} = p_g$

To optimize this measure encourages seeking only some modes of the data if the generator is not expressive enough. This means that the discriminator can get very confident about the generated samples, this combines with the activation function that computes the output of the discriminator. In this case, given that we have a binary problem, the output is computed by a sigmoid function. As we can see on figure 10.2, on the first column we have the sigmoid activation function and its derivative, the second column corresponds to the loss that is passed to the generator that is computed from the discriminator for he generated samples. If the generator is very confident on the classification, then the derivative of the activation function gets very sharp, and the loss that is passed to the generator ($\log(1-D(G(x)))$) gets basically zero for any sample. This means that the information that is passed to the generator about how to create better samples can not guide it towards this goal.

One possible solution is to be careful about how many optimization steps are done for the discriminator each iteration, as we have seen that is done on the original GAN training algorithm, but this is hard to decide and control. This would be another hyper-parameter that will have to be experimented and tuned.

One alternative that works well in practice is to change the loss of the generator, using instead:

$$\max_G \mathbb{E}_{z\sim p(z)}[\log(D(G(z)))]$$

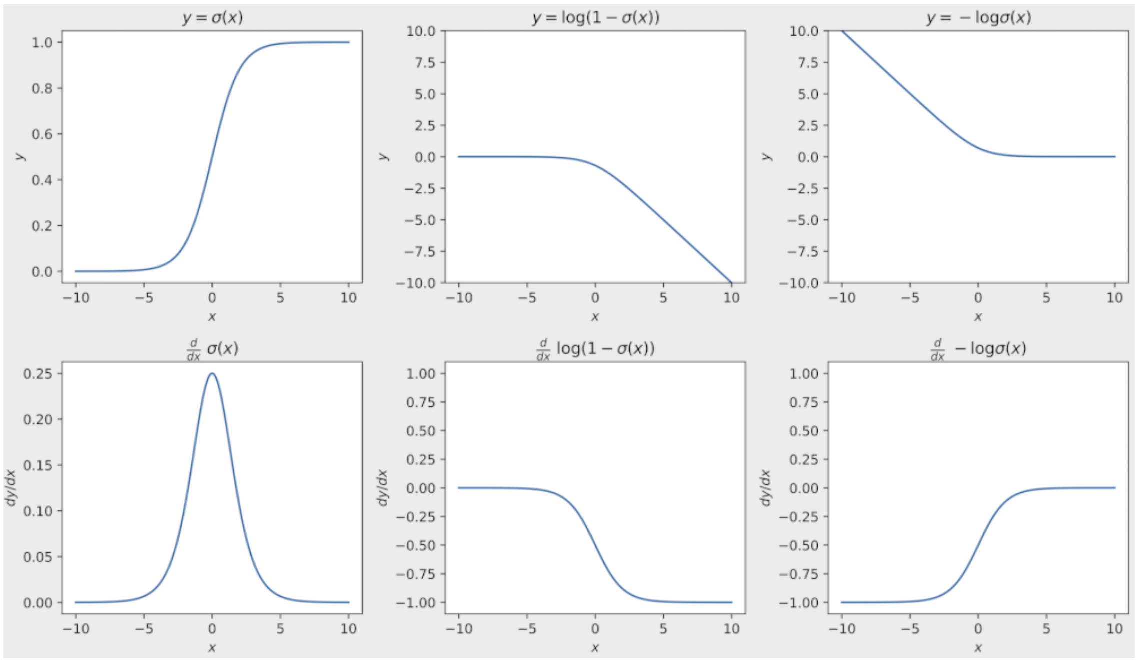


Figure 10.2: Sigmoid and the saturated and unsaturated loss for the GAN generator

This eliminates the saturation as we can see on the last column of figure 10.2. We are reversing the function that is used as loss for the generator, moving the non-zero gradients to the side of the generated images. This is actually not a problem in practice, since the generator does not use the gradients of the real images. Now the GAN optimization is no more a minimax game since the optimization of discriminator and generator are different, but given that it works better, this is the default used by most GANs that use this kind of loss. This GAN loss is called the unsaturated version of the original GAN loss:

$$\begin{aligned}\mathcal{L}_D &= \mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))]\\ \mathcal{L}_G &= \mathbb{E}_{z\sim p(z)}[\log(D(G(z)))]\end{aligned}$$

10.3 The evolution of GANs

It has been a lot of progress since the original definition of GANs. The main problems that present its training have been addressed with different choices on the architecture of the generator and the discriminator and choices about the training and the losses used for optimization.

Most of the advances have been done for models that generate images, but they usually extrapolate to other kinds of data. In this section we are going to review the different successful architectures that have been developed as a way to gain some insight about how these models work.

10.3.1 DCGAN

DCGAN [110] was the first GAN network able to generate images with an acceptable quality. This was the result of many experiments about the different choices that can be taken when defining the architectures of the generator and discriminator, so it could be used as a do and do not list for training GANs. One of the things that wanted to be tested was if all the tricks of the trade of supervised CNN architectures for object recognition were also valid for GANs.

Usually the basic architecture for the generator based on CNNs consists of up-sampling

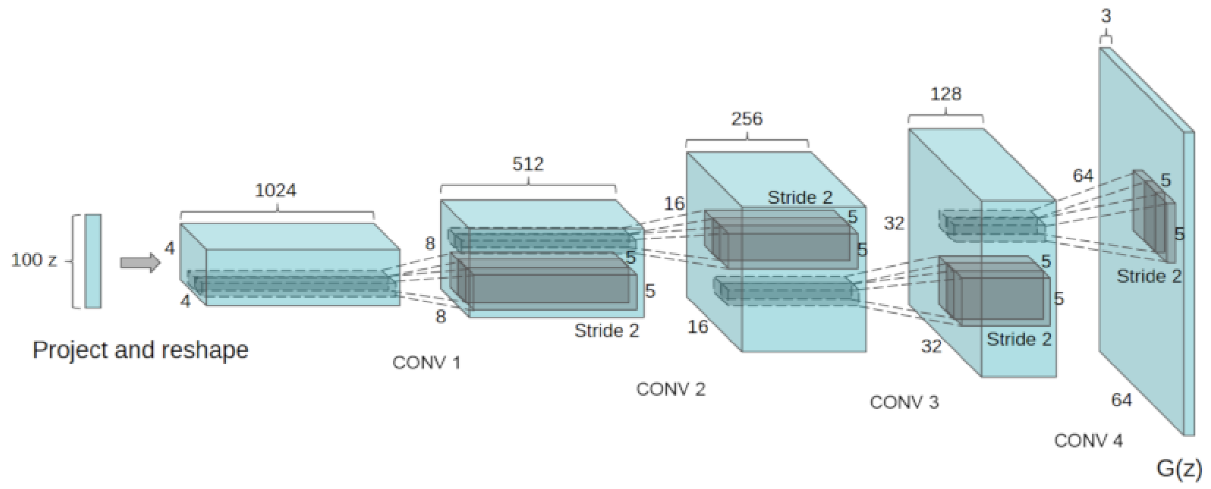


Figure 10.3: Architecture of the generator of DCGAN (source [110])

layers that transform a latent vector to an image. The discriminator usually follows a symmetrical architecture, performing down-sampling until reaching the classification layer. This is basically the architecture used for DCGAN, but with many specific choices. Figure 10.3 shows the general architecture of the generator.

One of the lessons of this GAN is not to use max or min pooling after the convolutional layers (something that is common on supervised networks) and to perform the upsampling and downsampling differently on generator and discriminator. The upsampling in the generator uses transposed convolutions, meanwhile, the discriminator uses strided convolutions combined with average pooling. The activation functions also need to be different on both networks, in this case ReLU was used for the generator and LeakyReLU for the discriminator.

The output activation function that worked better was *tanh* for the generator and *sigmoid* for the discriminator. Mode collapse was alleviated by using Batch normalization on the layers, but not in the output of the generator and the input of the discriminator.

The final architecture needs many hyperparameters that have to be tuned and different choices could lead to unstable training or mode collapse. The results depend on many specific choices, so this is a very brittle architecture that is hard to be trained.

When the choices are right, the generator is able to approximate reasonably well the data distribution (you can check some samples in the original paper).

Given that there is no specific benchmarks for generative models, many of the experiments are performed using datasets with specific sets of images or well known object recognition datasets, that allows visual evaluation. It is typical to use faces datasets like CelebA or parts of the LSUN dataset that includes for example images of churches or bedrooms. Apart from visual inspection most papers use the Inception Score and the Fréchet Inception Score as we explained on a previous section. This paper does not use these measures since it pre-dates their definition. This leads to measuring the capability of the generator in more eclectic ways.

One of these ways was to check if the latents corresponding to different images can be interpolated obtaining a continuous transition from one image to another. If this is the case, the distribution learned by the model is probably continuous rather than several isolated clusters of probability.

Another experiment that was performed is to check if the generator had some semantic understanding of the content of the images. On language embedding methods it is typical to perform arithmetic operations between words adding and subtracting their embeddings for

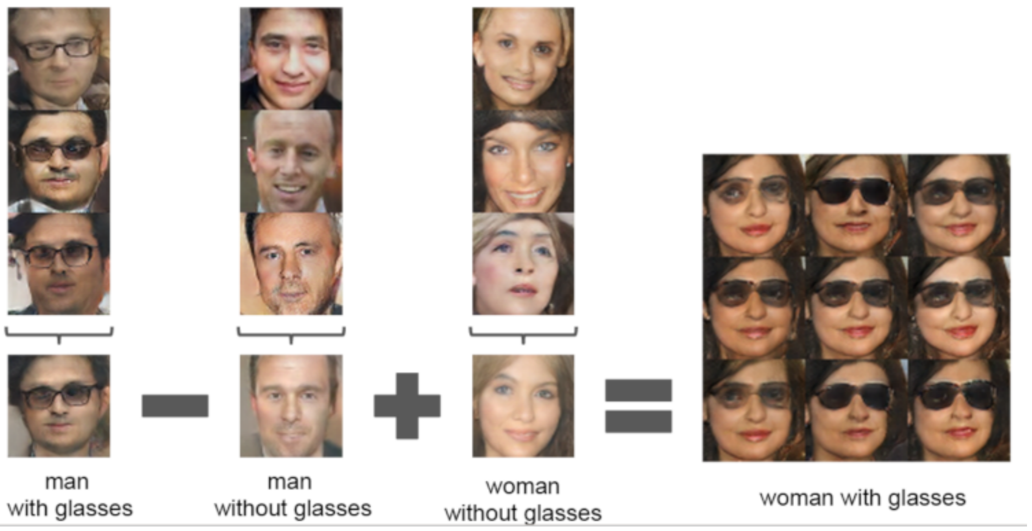


Figure 10.4: Arithmetic operations in the latent space learned by DCGAN (source [110])

obtaining new embeddings values close to words coherent to the operation. For instance, on word2vec embeddings, performing the operation (king - man + woman) obtains the embedding for queen. In the experiments, they train generators focused on data with specific attributes and identified the latents that generated images with these attributes using a classifier. These latents could be operated with to obtain similar effects as in word embeddings. In figure 10.4, you can see images where from the latent of a man with glasses is subtracted the mean latent of men without glasses and added the mean latent of women without glasses to obtain samples of women with glasses. This shows that some kind of disentanglement is happening inside the network that detects the target attributes.

10.3.2 Wasserstein GAN

The cross entropy of the real and generated data is not the only way of measuring how close the two distributions are. One alternative is the **Wasserstein Distance** or Earth Move Distance that is used in optimal transport. In the domain of probability distributions this distance measures the cost of moving the probability mass of one distribution, so it is transformed in the other. The goal is to define an optimization problem that obtains the optimal distance between a pair of distributions, this can be defined as:

$$W(P_a, P_b) = \inf_{\gamma \in \Pi(P_a, P_b)} \mathbb{E}_{(x,y)\sim\gamma}[||x-y||]$$

Where $\Pi(P_a, P_b)$ represents the set of distributions $\gamma(x,y)$ whose marginals are P_a and P_b. It is obviously intractable to search in the space of all possible joint distributions to obtain a solution. There is a mathematical result named the Kantorovich Rubinstein Duality that allows rewriting this formulation as:

$$W(P_a, P_b) = \sup_{||f_L|| \leq 1} \mathbb{E}_{x\sim P_a}[f(x)] - \mathbb{E}_{x\sim P_b}[f(x)]$$

Where the search is defined over 1-Lipschitz functions. A function is K-Lipschitz if for distance functions d_x and d_y

$$d_y(f(a), f(b)) \leq K \times d_x(a,b)$$

.

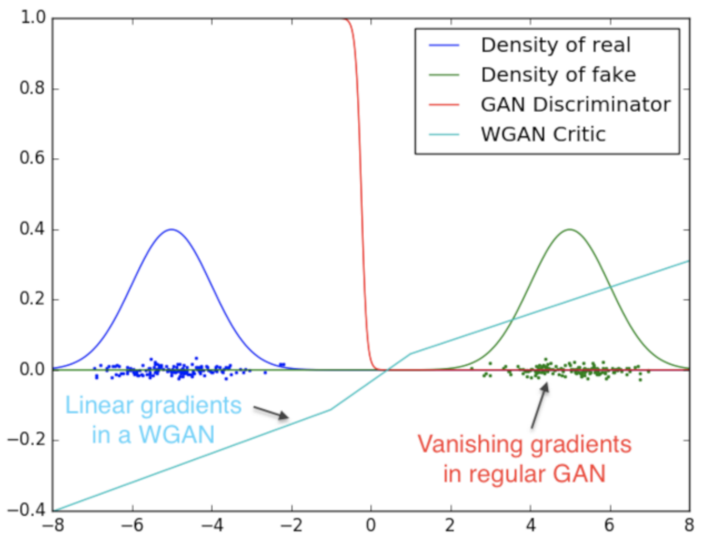


Figure 10.5: Difference of gradients between Wasserstein distance and cross entropy

This means that in all the domain of the two functions the difference between their image values never exceeds a factor K of the distance between the values. This poses a constraint over the smoothness of the function. Basically we want to obtain functions that interpolate smoothly between any two points.

This is also intractable, since there are infinite many 1-Lipschitz functions, but we could search over a set parametrized functions f_w that is a lower bound of these functions, so:

$$\max_w \mathbb{E}_{x\sim P_a}[f_w(x)] - \mathbb{E}_{x\sim P_b}[f_w(x)] \leq \sup_{||f_L||\leq K} \mathbb{E}_{x\sim P_a}[f(x)] - \mathbb{E}_{x\sim P_b}[f(x)] = K \times W(P_a, P_b)$$

We can define this set of parametrized functions as a neural network, and that is a problem that we can optimize using gradient descent.

This is what is applied by the **Wasserstein GAN** (WGAN [6]) that uses the loss:

$$\min_G \max_D \mathbb{E}_{x\sim P_D}[D(x)] - \mathbb{E}_{\tilde{x}\sim P_G}[D(\tilde{x})]$$

As you can see, the discriminator acts as the measure that defines how close are the distributions of the real and generated data. In this case this network is no longer called a discriminator, but a **critic network**. It evaluates the similarity of the distributions and not if the examples are real or generated.

One interesting observation is that the Wasserstein distance experimentally correlates with the quality of the images computed with IS and FID, so it is able to help the generator to improve. Also, the gradient obtained with this distance is more informative eliminating the vanishing gradient that is a main concern when using the cross entropy as loss. This is represented on figure 10.5, where the cross entropy loss saturates and gives the same gradient information for all the samples meanwhile the Wasserstein loss depends on how distant are the samples to the distribution of the real data.

WGAN has also other advantages with respect to DCGAN, it can obtain results of similar quality for different architectural or training choices, and it is more stable.

Despite the success of WGAN, the original implementation uses a weird way for forcing the neural network to be a 1-Lipschitz function. It simply clips the weights of the critic network to be inside a range of values. This leads to a variant with a more justified solution for maintaining the smoothness of the computed function using regularization. In this case, an additional term

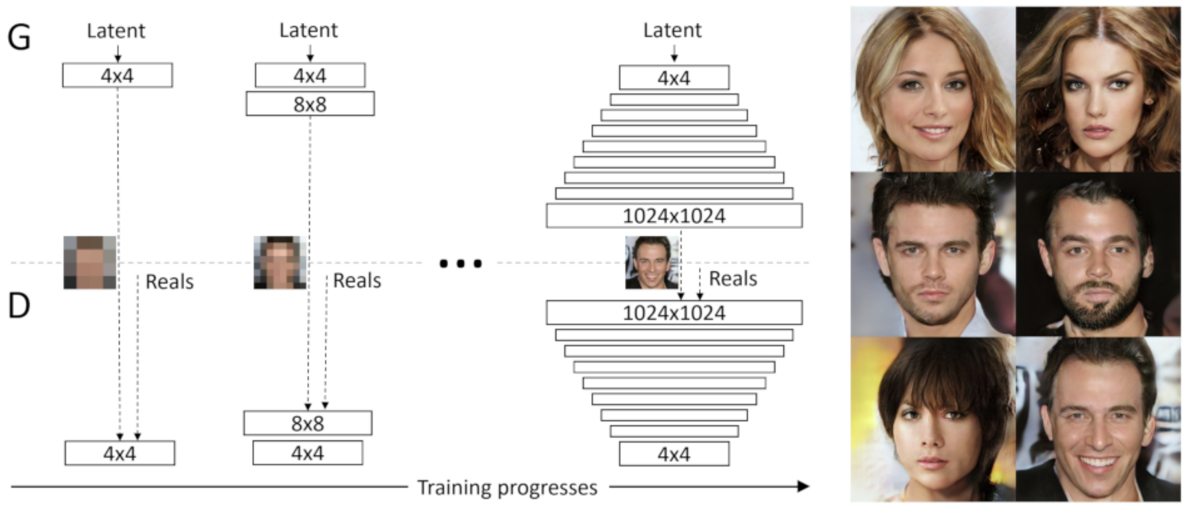


Figure 10.6: Architecture and training of Progressive GAN and samples at 1024 pixels resolution (source [76])

is introduced in the loss function that corresponds to a penalization if the gradients of the critic computed function are away from one. This defines the **Wasserstein GAN with Gradient Penalty** (WGAN-GP [56]). This is justified by the observation that a 1-Lipschitz function must have gradients with norm at most one everywhere.The modified loss function is:

$$\mathbb{E}_{x\sim P_D}[D(x)] - \mathbb{E}_{\tilde{x}\sim P_G}[D(\tilde{x})] + \lambda\mathbb{E}_{\tilde{x}\sim P_{\tilde{x}}}[(||\nabla_{\tilde{x}}D(\tilde{x})||_2 - 1)^2]$$

This weight is computed by sampling points in a straight line between samples from P_D and P_G. If the gradient on these intermediate values is not around one then the function is not smooth enough.

This loss has been adopted by many GANs, but has some drawbacks, it is more costly to compute, convergence could be slow, and the penalty is still computed heuristically.

10.3.3 Progessive GAN

Progressive GAN [76] is the first GAN to obtain high resolution images. All the previous models had problems with images of resolutions larger than 256 pixels. This GAN was able to achieve good results up to 1024 pixels. The main problem of previous GANs is that with higher resolutions it is easier to have artifacts on the image produced by the up-scaling. These flaws were easily detectable by the discriminator and the generator could not obtain enough guidance for correcting them.

The solution of Progressive GAN was to train a succession of models that increased the resolution each new step. Each new model bootstraps on the previous one. First a GAN is trained for a specific resolution, its weights are frozen and a new layer that increases the resolution is added.

Generator and discriminator are symmetrical networks as can be seen on figure 10.6. Each increase on image resolution in the generator adds a residual connection between the trained layers and the new layer. The output of the last lower resolution layer is up-scaled using quadratic interpolation and a convex combination of this up-scaling and the output of the new layer is used as output. The weight of the convex combination is decreased progressively down to 0 during training. This maintains the stability of the new training phase obtaining a reasonable output from the beginning. For the discriminator, the images are down-scaled interpolating between resolutions similarly. The training also normalizes the feature vectors for the pixels to unit length to constraint the values in the discriminator and generator. Some results at a resolution of 1024 pixels are shown in figure 10.6.

10.3.4 Spectral Normalization GAN

Spectral Normalization GAN [96] continues with the idea of forcing the function computed by the discriminator to be smooth. In this case with a more theoretical justification. The basic idea is that the smoothness of the function computed by a layer of a neural network depends on the spectral norm of its weights. The spectral norm of a matrix is defined as:

$$\sigma(A) = \max_{||x||_2 \neq 0} \frac{||Ax||_2}{||x||_2}$$

The solution to this problem corresponds to the first singular value of the matrix.

Since a neural network is composed by functions defined by each layer, each parameterized by a matrix of weights W, where there is also possibly a non-linear activation function, the function computed by the whole network can be defined as:

$$f(x,\theta) = W^{L+1} a_L(W^L(a_{L-1}(\cdots a_1(W^1 x)\cdots)))$$

By definition, the Lipschitz norm is:

$$||g||_{Lip} = \sup_h \sigma(\nabla g(h))$$

For a linear layer $g(h) = Wh$ the norm will be $\sigma(W)$. Thus, if the Lipschitz norm of the activation functions is equal to 1 (for instance, it is for ReLU and Leaky ReLU, and some others are K-Lipschitz) we can use the inequality:

$$||g_1 \circ g_2||_{Lip} \leq ||g_1||_{Lip} \cdot ||g_2||_{Lip}$$

approximating the norm of the network as:

$$\begin{aligned} ||f||_{Lip} &\leq ||W^{L+1}h_L||_{Lip} \cdot ||a_L||_{Lip} \cdot ||W^L h_{L-1}||_{Lip} \cdots ||a_1||_{Lip} \cdot ||W^1 h_0||_{Lip} \\ &= \prod_{l=1}^{L+1} ||W^l h_{l-1}||_{Lip} = \prod_{l=1}^{L+1} \sigma(W_l) \end{aligned}$$

This means that if for all the layers the matrix parameter W is normalized by the spectral norm $\sigma(W)$, then the function that is computed by the network will be bounded by norm 1.

To compute the eigenvalues of a matrix is expensive. If we want to obtain the exact values this involves computing its SVD. This computation scales $O(n^3)$, and it is not feasible for large networks. To solve this problem an approximation is used. Since only the first eigenvalue is needed, for any matrix this can be approximated using the **power iteration method**. This consists on starting with a random unit vector that is multiplied by the matrix and then normalized again, repeating this operation several times converges to the real value. We are performing this update:

$$v_{k+1} = \frac{Wv_k}{||Wv_k||}$$

That allows computing the first eigenvalue after a number of iterations as:

$$\lambda = \frac{Wv_k \cdot v_k}{v_k \cdot v_k}$$

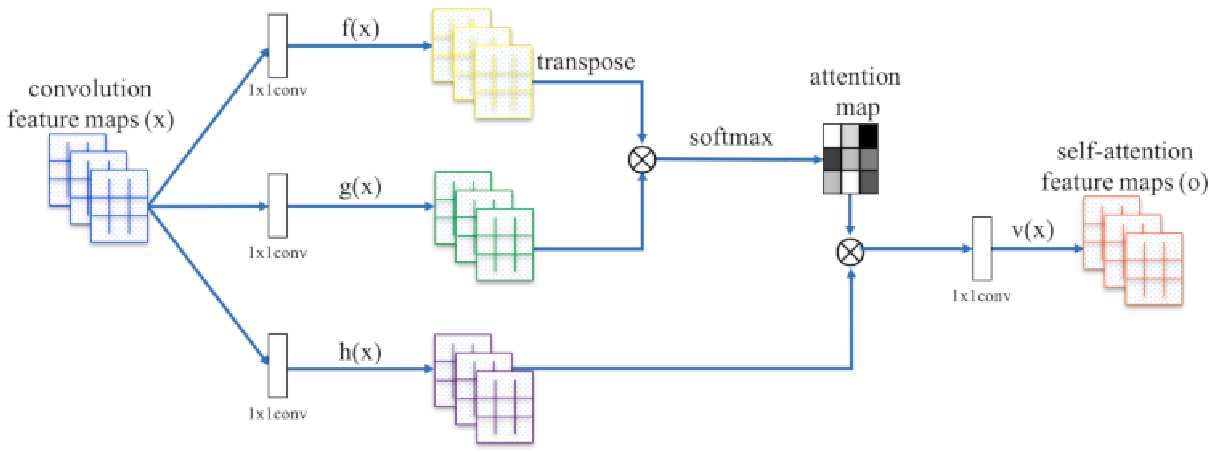


Figure 10.7: Architecture of Sel Attention GAN (source [141])

This is still expensive depending on the number of iterations, but in practice just one iteration should obtain an approximation that gives good results.

Spectral Normalization GAN also introduces a change on the loss function of the discriminator using the hinge loss (the one from SVMs) instead of the cross entropy. We are optimizing:

$$V(G,D) = \mathbb{E}_{x\sim p_{data}}[\min(0, D(x)-1)] + \mathbb{E}_{z\sim p(z)}[\min(0, -1-D(G(z)))]$$

10.3.5 Self Attention GAN

Self Attention GAN [141] introduces self attention [131] in the generator and discriminator networks. This works extends on Spectral Normalisation GAN extending the spectral normalization with attention weighting.

The self attention is applied to increase the information that is used by the network over the local information generated by the convolutions.

The attention layer is applied to the convolution filters using them as key, value and query. First the filters are passed through 1x1 convolutions applying the usual operations of self attention, key and value are multiplied and softmaxed and that is multiplied with the query, the result is passed through 1x1 convolutions. Figure 10.7 shows these operations.

The self attention is added to the original filters multiplied by a scalar that is a learnable parameter. It is initialized to 0, so it allows learning first how to use the local features and then includes the information of the self attention.

The discriminator uses the hinge loss, but the generator uses the non-saturated version of the classical loss.

Spectral normalization is used in the discriminator and the generator to stabilize the training and also different learning rates are used for the networks to let the discriminator train faster, instead of doing more iterations for the discriminator as in other GANs.

10.3.6 BigGAN

BigGAN [15] is the first success for an architecture that generates high resolution images without needing progressive training. It also uses a modified ResNet architecture, opening the way to using architectures commonly used for object classification.

This network combines many of the lesson learned by the previous models including spectral normalization, self attention and the use of the Hinge loss. Other lessons can also be learned from this network, for instance, that the depth and the number of filters on each convolutional

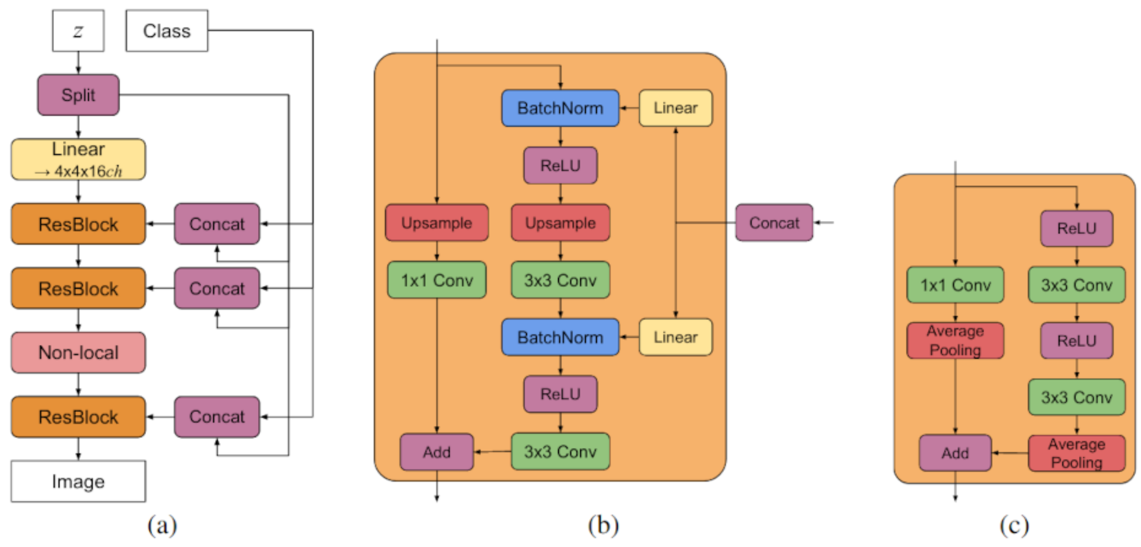


Figure 10.8: BigGAN Generator architecture (a) and residual blocks for the genrator (b) and the discriminator (c) (source [15])

layer of the network are important for the quality of the samples and also the size of the batches used when training, hypothetically because more modes are included on each epoch and also better gradients can be computed.

This network adds also an additional regularization that penalizes non-orthogonal weight matrices but excluding the diagonal elements. This adds to the loss

$$||W^\top W - I||_F^2 \Rightarrow ||W^\top W \odot (1 - I)||_F^2$$

Figure 10.8 shows the general architecture of the generator. It is composed by convolutional residual blocks that perform the upscaling of the latent vector. It also has to be noted that this network can be conditioned, meaning that an additional information vector can be introduced on the residual block that corresponds to the class of the image we want to generate. An additional trick that is used on other GANs is to split the latent vector in several sub-vectors that are used to assign different parts of the latent to different scales. Also in figure 10.8 can be seen the difference between the residual blocks in the generator and the discriminator. The blocks on the discriminator are basically the operations that are performed by ResNet. The blocks on the generator use the Batch normalization to introduce the conditioning before an after the scaling.

There is not only one BigGAN, there are different variations that scale the number of parameters for better results and each introduce slight variations. This network is costly to train since it is usually larger that the memory available on current GPUs and also needs very large batches. This means to use a training that distributes the model and also the data.

An additional trick introduced by this network is to perform the sampling truncating the values on the gaussian latent. This is called the **truncation trick**. The idea is to define a maximum value for the latent variables and, when generating the latents, all the values that exceed the threshold are changed to this maximum value. This improves the quality of the samples, but decreases their diversity. Figure 10.9 shows some of the samples obtained by this model when trained with the Imagenet dataset.

10.3.7 StyleGAN

StyleGAN [75] is the first model that departs of the usual generator architecture. This model introduces two networks for the generation of samples. The first one performs the up-scaling

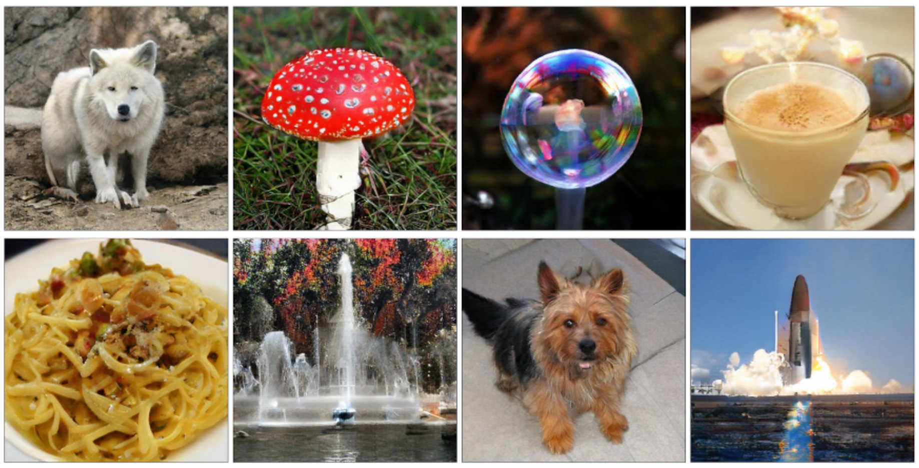


Figure 10.9: Samples generated by BigGAN (source [15])

from a latent, that for this model is fixed. This is called the **synthesis network** and it is inspired by the progressive GAN, but it is trained in the traditional way, targeting the maximum resolution. The second network is a fully connected network that transforms a gaussian latent. This is called the **mapping network** and aims to control what is generated by the other network. The output of this network is called a **style code**. This is introduced on each of the up-scaling blocks of the synthesis network. Additional noise is introduced at the same time on each up-scaling to improve generation quality. Figure 10.10 shows the architecture of these two networks compared to a traditional generator. The discriminator is the same one as in Progressive GAN.

The key point of this architecture is the called **adaptive instance normalization**. The feature maps of each upscaling block are translated and scaled using the information of the style code, so the changes in the generation of each scale depends directly on its information. This is different from other normalization techniques that are used in neural networks that normalize the layers or the activations using statistics gathered from the whole dataset during training. The noise is different for each scale and controls the stochasticty of the generation.

One interesting thing observed in this network is that it is able to perform some disentangling of the latent attributes of the data and different resolution scales control different things in the generated images. The first/middle resolutions are able to control what is the content of the image through the style code and the higher resolutions control the finer detail. It is possible even to mix different style codes to transfer style information from an image to another. Taking the style code that generates and image and inserting it on different layers instead of the style code of other image modifies the output attending to the content/style of the introduced code. Figure 10.11 shows this effect, the source A images are changed with the style codes that generate the source B images changing the code of image A for the one of image B at different levels of the synthesis network. Coarse levels can control for instance gender, age, pose, smile or facial elements like glasses, middle levels change face shape for instance and finer levels perform more cosmetic changes like hair color.

This network was improved on StyleGAN2 [77] since it was observed that for some generated images had artifacts that made easier for the discriminator to detect them. These artifacts were caused by the way the normalization was performed and how the images were upscaled to the different resolutions. The paper shows different variations of the normalization changing where the style codes and the noise are introduced in the architecture and the kind of transformations that are used to change the feature maps from the convolutional layers.

Figure 10.12 shows details of the differences between the original instance normalization of StyleGAN. It can be seen in (c), on a first revision of the network that now the noise is introduced

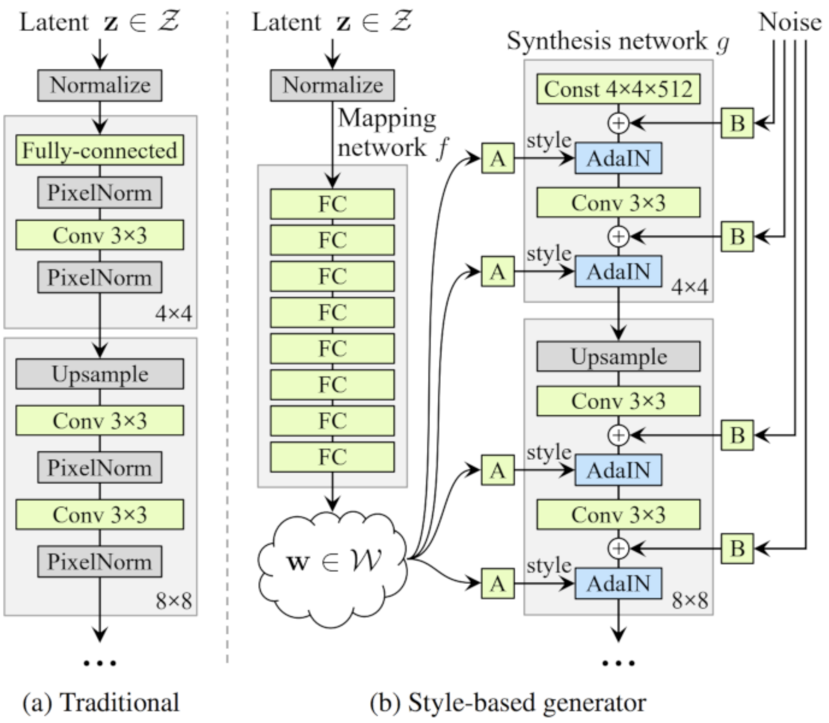


Figure 10.10: StyleGAN architecture (source [75])

after each upscaling block. The normalization using the style codes now uses only the standard deviation instead on the mean and the standard deviation. This is finally changed in (d) by a set of weight that are computed from the style codes and a learnable set of weight that are transformed using what are denominated modulation and demodulation operations. These operations scale first the weights using the style codes and then downscale them using the standard deviation of the whole set of weights. This ends being a set of weights that are used to scale the weights of the feature maps of the convolutions. Figure 10.13 shows some outstanding samples from this network trained with the Flickr Faces High Quality (FFHQ) dataset.

One of the problem that still has this network (and any network that is based on convolutions with border padding) is that it is not translation and rotation invariant, so navigating in the latent space maintains textures in place. The source of the problem is that the discretization of the continuous signal that is the real image leads to interpolations that break the equivariance to translation and rotation when convolutional filters are used. Basically when processing the images it appears in the signal frequencies that are higher than the Nyquist frequency of the original image (more than half the higher frequency in the signal).

A set of modifications were performed for the StyleGAN3 architecture that eliminates that problem and does not reduce the FID score of the generated images. The modification involves how convolutions are performed and how the image is up-sampled and down-sampled. Also, band pass filters are applied to cut the frequencies higher than the frequency limit allowed by the resolution of the image.

The architecture has many variants that perform several changes, among them:

- Extending the border of the image, so there is no padding, and it has less effect in translation invariance
- Performing up-sampling before the Leaky ReLU activation and then down-sampling to the current resolution

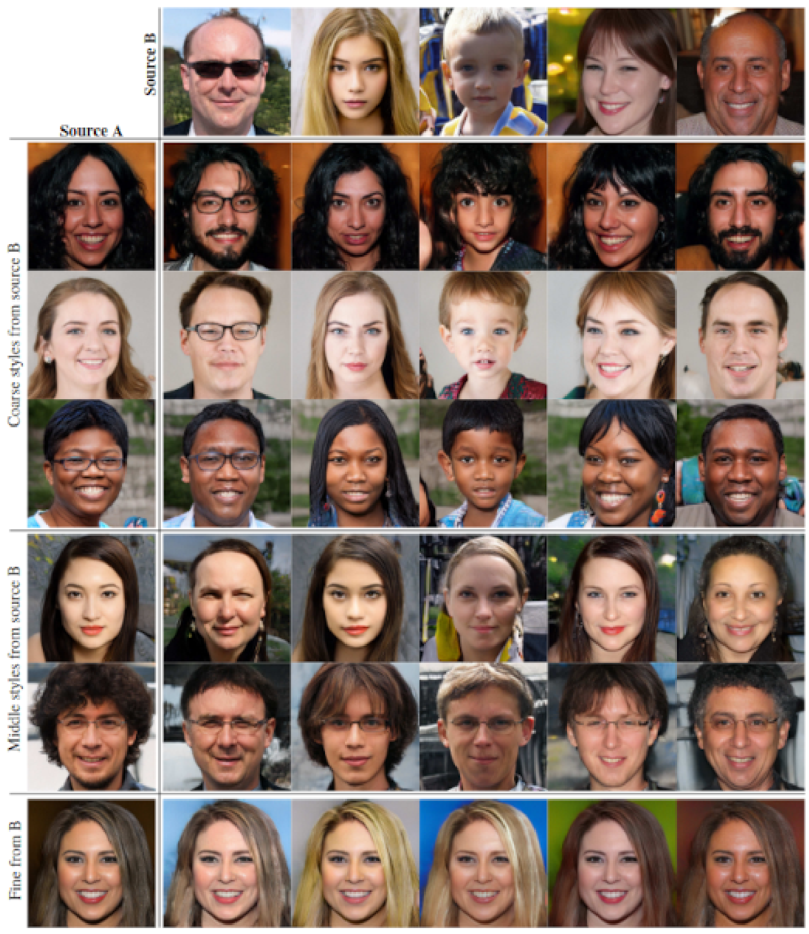


Figure 10.11: Mixing style codes from different images at different layers (source [75])

- Changing the initial fixed latent used by the synthesis network that corresponds now to Fourier features, improving the range of frequencies that are used in the generation
- Adjusting the frequency cutoffs and the resolution increase layer per layer
- Changing the 3x3 convolutions to 1x1 convolutions and the number of filters used for the down-sampling
- Eliminating the induced noise that is introduced in the network for some of the layers given that it is not translation invariant

The advantage of this network is that images can be translated manipulating the input latent and the network is able to learn a kind of coordinate system that helps to perform translations and rotations. The main drawback is that all these modifications increase the number of parameters of the network doubling the training time.

10.4 Conditioning

As we have seen in BigGAN, some models can be conditioned so they generate samples that are associated to an auxiliary value. This is not limited to labels, we can introduce different elements as part of the input in generator and discriminator that correspond to what has to be used to change the content of the generator. This can include for instance other images, allowing

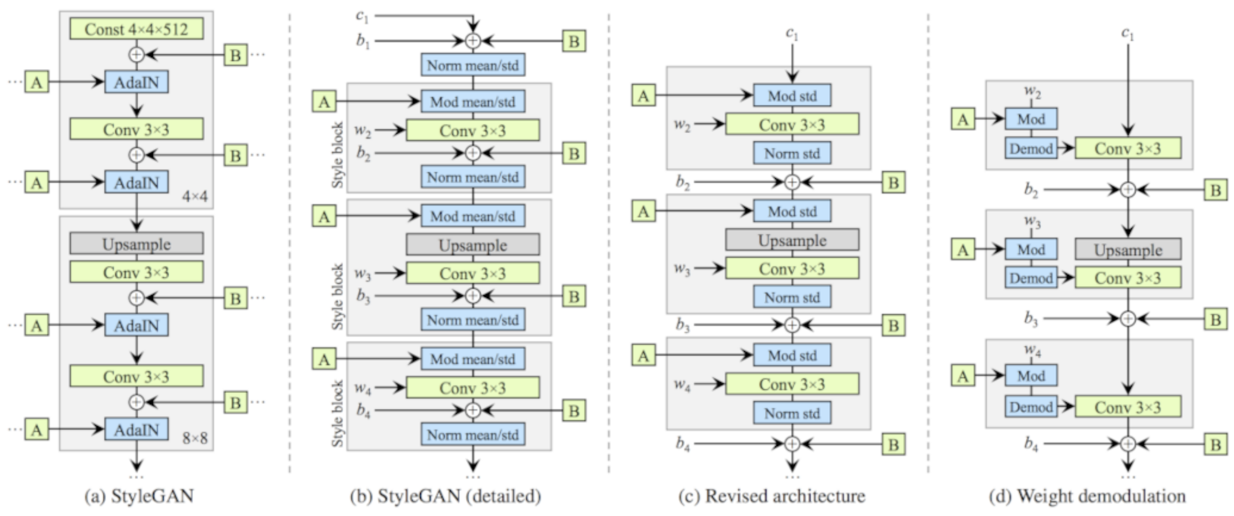


Figure 10.12: Changes on the original instance normalization of StyleGAN for StyleGAN2 (source [77])

to obtain for example image to image translation or to use text for obtaining a model that is a text to image generator.

There are different ways for performing the conditioning of a GAN. In figure 10.14 there are shown four variations. The original cGAN [94] concatenated the additional information to the input latent vector, this is effective if the information corresponds for instance to a class or other simple information. This was modified by [112] introducing the information after the latent has been transformed to a more complex set of features by the first layers of the generator where it is easier to concatenate a more complex conditioning, for instance an image or a mask.

To further improve class conditioning Auxiliary Classifier GAN (ACGAN) [102]) introduced an auxiliary classifier additionally to the discriminator. The loss from the classifier is added to the loss of the discriminator to guide the generator. This has different variations where the discriminator and the classifier could be separated, they could share parameters except for the last layers or the fake class could be an additional class for the classifier.

The model cGAN with projector discriminator [95] uses a different approach using an embedding of the additional information that is multiplied using the inner product with the output of the last layer of the discriminator before the classification layer. This inner product is combined with the logits of the classification layer before computing the softmax. This modified output now depends on the conditioning and is used to compute the loss that is backpropagated to the generator.

10.4.1 Pix2pix

Pix2pix [72] is a model that is able to perform image to image translation. This allows many applications like generating images from sketches, changing the style of an image or to translate from different image modalities like greyscale to color.

In this case the dataset consists of paired images, the source image and its translation. Instead of a latent vector, the generator network receives an image that corresponds to the source of the translation, the goal is to obtain the corresponding translated image. The generator network is an encoder decoder based on the U-Net architecture. The discriminator now has to distinguish if the pair of images come from the training dataset or is a pair of real/generated images (see figure 10.15).

Figure 10.13: Samples generated by StyleGAN2 (source [77])

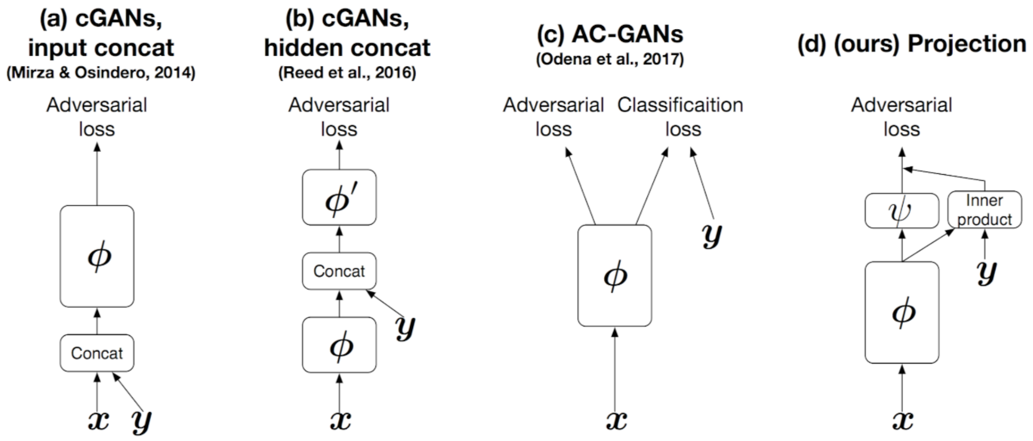


Figure 10.14: Types of GAN conditioning (source [95])

This translation is challenging for the GAN loss since, for instance, if we want to colorize images, any image that has color and is real enough would fool the discriminator. We need the images as close as possible to the real images. This can be achieved by penalizing the distance between generated and real translation. For this, the loss of pix2pix used the unsaturated GAN loss plus a penalization that computes the L_1 distance between the images.

Still this is not enough to capture the fine details of the images, so this model introduces a new discriminator that works at patch level. The input image is divided on patches and the discriminator has to decide for each patch if it corresponds to a real image or to a generated image, obtaining as many decisions as patches. This allows to match the finer details of the images. This is known as **Patch GAN** and has been used after this by many other GANs.

10.4.2 CycleGAN

One of the drawbacks of image to image translation is the need to obtain a dataset of paired images, this is not always possible. CycleGAN [146] proposes a model that can perform

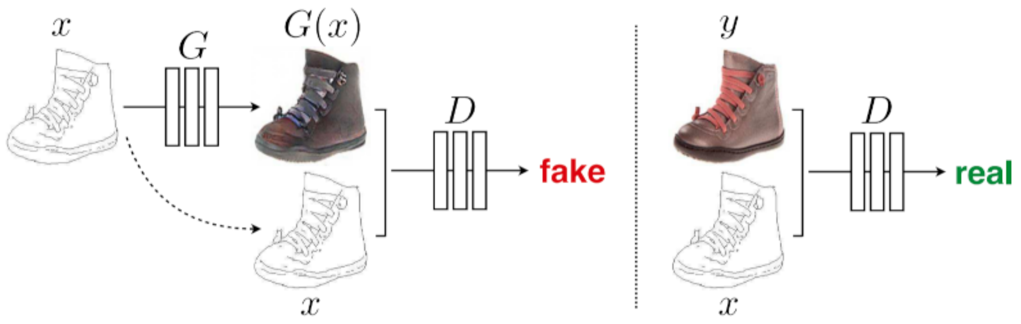


Figure 10.15: Pix2pix discriminator for real/real and real/generated images (source [72])

translation without this constraint. The idea is to obtain mappings able to translate between two datasets of images. This can be achieved by maintaining the consistency between the results of both mappings. We want to force that the image generated by the first mapping obtains again the original image applying the second mapping. This is defined as cycle consistency:

$$Image_x \rightarrow G(Image_x) \rightarrow F(G(Image_x)) \rightarrow Image_x$$

Basically, we want a mapping that is able to translate an image to another domain (F) and when applying another mapping (G) the translation generates the original image.

This is obtained by training two adversarial models where their generators will correspond to the two mappings. Each GAN optimizes the usual adversarial loss, one generator from one GAN translates images from domain X to domain Y and its corresponding discriminator learns to distinguish images in domain Y and the other performs the same operation reversing the roles:

$$\begin{aligned}\mathcal{L}_{GAN}(G, D_Y, X, Y) &= \mathbb{E}_y[\log D_Y(y)] + \mathbb{E}_x[\log(1 - D_Y(G(x)))] \\ \mathcal{L}_{GAN}(F, D_X, Y, X) &= \mathbb{E}_x[\log D_X(x)] + \mathbb{E}_y[\log(1 - D_X(G(y)))]\end{aligned}$$

In order to force the matching of the images when mapping from one domain to the another and back, a loss based on the $L1$ distance is added to the adversarial losses. This corresponds to the cycle consistency loss:

$$\mathcal{L}_{cyc}(G, F) = \mathbb{E}_x[||F(G(x)) - y||_1 + \mathbb{E}_x[||G(F(y)) - x||_1]$$

This completes optimization goal, where an additional weight (λ) controls how much is forced the faithful reconstruction of the images:

$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_{GAN}(G, D_Y, X, Y) + \mathcal{L}_{GAN}(F, D_X, Y, X) + \lambda\mathcal{L}_{cyc}(G, F)$$

We can interpret this model as training two autoencoders at the same time that map the input they receive from one domain through an intermediate representation that corresponds to the encoding of images of the second domain.

This kind of unsupervised translation works better when the domains have similar characteristics and will be more difficult when the two domains are very different. Figure 10.16 shows samples obtained with this GAN for some domains including style transfer from photographs to paintings, changing the season of a photograph or translating from horses to zebras.

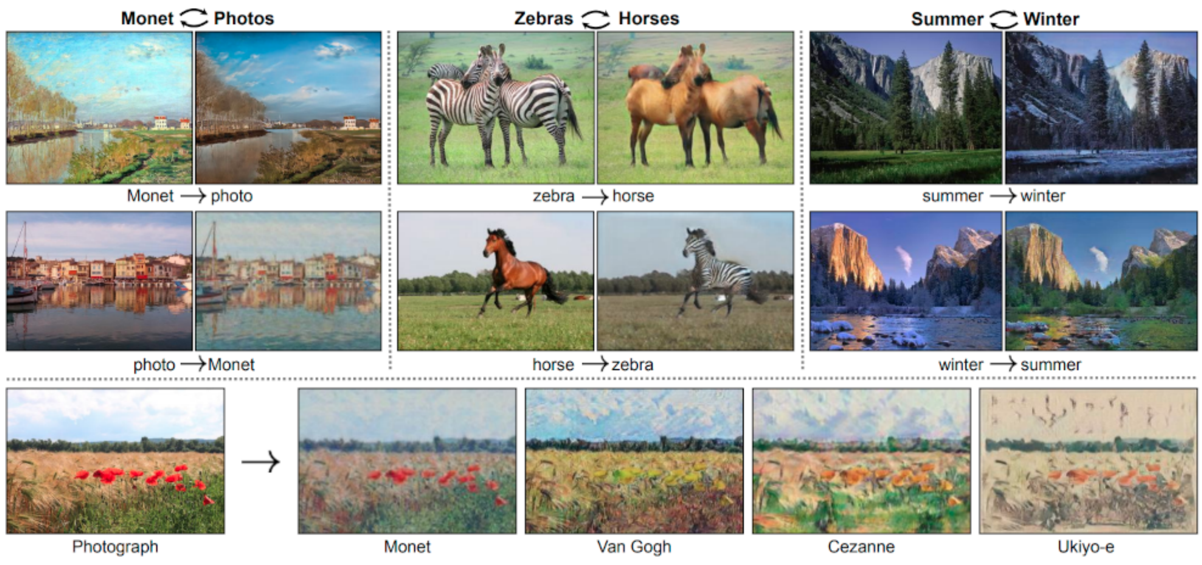


Figure 10.16: Image to image translation without paired images using CycleGAN (source [146])

10.5 Feature Learning/Disentanglement

As we explained on the VAEs chapter, some models are able to perform disentangling of the features that generate the data. This can be done in different ways. We are going to introduce two networks, one that learns to associate specific latents to attributes of the samples and other that learns an encoding network that can be used for other tasks.

10.5.1 InfoGAN

InfoGAN [23] assumes that we have a domain where we know different attributes that control the characteristics of the sample. The assumption of this model is that the gaussian noise that is the input of the generator is coding this information but in an entangled way. The goal is to factorize the input latent on an entangled part (the gaussian noise z) and a structured part (controlling variables c). The structured part could correspond to continuous or discrete variables that will be learned at the same time that the GAN is trained.

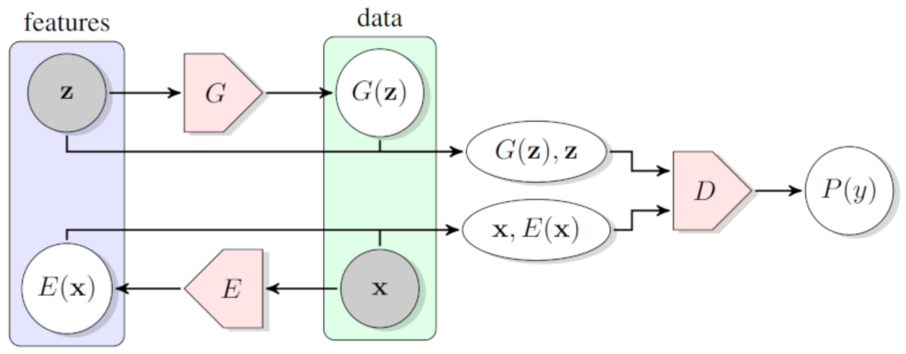
We define the structured part of the input choosing a set of variables that will code the attributes we want to learn. This is slightly different from conditioning using the class of the examples since we can use a different number of discrete or continuous variables that will associate to characteristics of the samples.

To force the generator to use the structured information of the input of the generator, both parts are linked using a term in the loss that computes the mutual information between them, so the loss is better if one part is able to predict the other. This means that we are correlating the information from the structured variables and the noisy latent. This mutual information corresponds to:

$$I(c;G(z,c)) = H(c) - H(c|G(z,c)))$$

To perform this computation directly is intractable, so Variational Inference is used learning a distribution that approximates the generator distribution. A neural network is used to approximate the distribution and the ELBO is used as optimization objective. To reduce computational cost, the network for the Variational distribution shares its parameters with the discriminator of the GAN. The complete loss corresponds to:

$$\min_{G,Q} \max_{D} V_{infoGAN}(D,C,Q) = V(D,G) - \lambda I(G,Q)$$



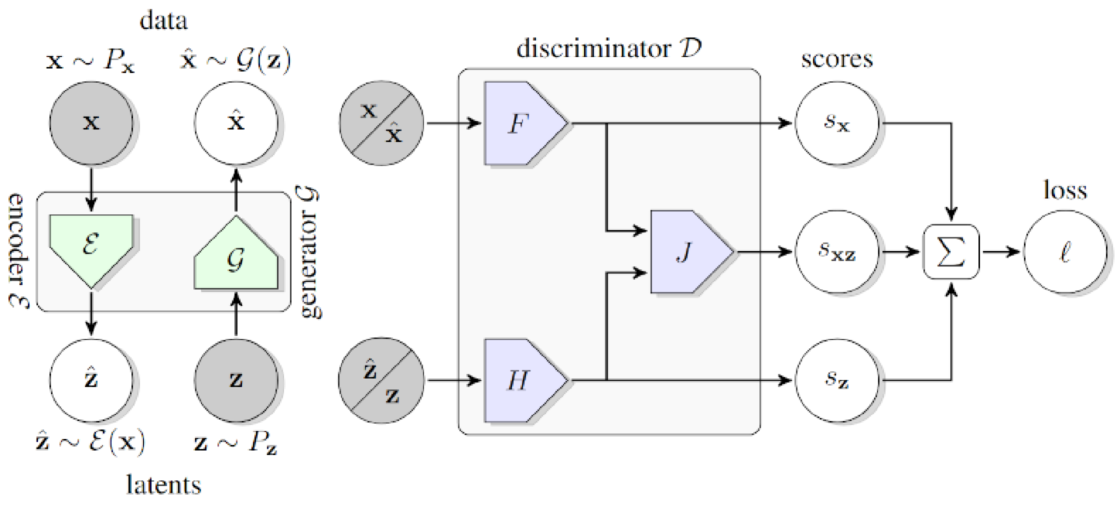


Figure 10.17: Architectures of Bidirectional GAN (up) and BigBiGAN (down) (source [36],[37])

Where V is the adversarial loss of the GAN and I is the mutual information computed with the variational distribution.

10.5.2 Bidirectional GAN

Bidirectional GAN (BiGAN [36]) adds an encoder to the training of the GAN for learning a network that obtains a latent representation of the samples. The optimization uses the code that the encoder network obtains from the real images and the discriminator has to be able to distinguish pairs (learned code, real example) from (noise, generated image). The loss is:

$$V(D,E,G) = \mathbb{E}_{x\sim p_x}[\log D(x,E(x))] + \mathbb{E}_{z\sim p_z}[1 - \log D(G(z),z)]$$

The effect is to learn a network that when optimized is able to invert the generator. This inverted code corresponds to the latent representation of the real images, so it can be used as features for other tasks.

Figure 10.17 (up) shows a representation of the architecture. During training, the gradient from the discriminator is passed to the generator and the encoder. This training could be interpreted as an auto encoder but with an adversarial training.

For many applications the network used for extracting features corresponds to a standard network, for instance a ResNet. BigBiGAN [37] uses a ResNet50 as encoder so when trained it can be used as feature extractor for other tasks. This network also targets larger images, so an improved discriminator is used as can be seen in figure 10.17 (down). Two separate networks extract features from the images (F) and from the latents (H) these are used to obtain two losses determining if they are real or generated. These features are then combined by another network

(J) to determine if the pairs are real or generated and finally the three outputs are combined to obtain the loss that will be used by the generator and the encoder.

10.6 Generative Adversarial NEtworks (GANs)

In this session we will use different types of GANs. As you will remember, this model obtains a network capable of generating by transforming a vector of Gaussian veriables into examples (generator) by training it against a network that is capable of determining whether what the network generates is a real example or not (discriminator).

The two networks are trained to optimize a minimax game where the discriminator receives information about how well it is deceiving the discriminator and the discriminate receives information about how well it is able to distinguish real examples from generated ones.

We will use the `mimicry` library that implements several GAN models. This library has already defined different generator and discriminator architectures and the training of the models and their evaluation.

10.6.1 DCGAN

We will start by training a GAN for the MNIST dataset. To avoid making the training too long, we will reduce the data set to the first 10,000 and create a data loader for them.

[6]:
```
dsize = 10000
mnist_trainset = datasets.MNIST(root='../../data', train=True,␣
 ↪download=True, transform=None)
train_dataset = (mnist_trainset.data[:dsize].reshape(-1, 1, 28, 28) / 127.)␣
 ↪-1.
train_labels =  mnist_trainset.targets[:dsize]

class MNIST(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.label = labels

    def __getitem__(self, index):
        x = self.data[index]

        return self.data[index], self.label[index]

    def __len__(self):
        return len(self.data)
dataset =  MNIST(train_dataset, train_labels)

mnist_dl = torch.utils.data.DataLoader(dataset, batch_size=512)
```

[7]:
```
import torch.nn as nn

from torch_mimicry.nets.dcgan import dcgan_base
from torch_mimicry.modules.resblocks import DBlockOptimized, DBlock, GBlock
```

Let’s first define the generator as a network with residual blocks that scales from a latent which we will transform from (7,7) to (28,28). Since the data has been normalized to the range

[0-1] we will use a sigmoid function so that the output is in that range.

```
[8]: class DCGANGeneratorMNIST(dcgan_base.DCGANBaseGenerator):
         r"""
         Resnet for the DCGAN generator for MNIST. Images are (1,28,18), we will␣
     ↪start
         from a transformation of the latent to an image (7,7) and we will make
         upscaling until the final size

         Attributes:
             nz (int): size of the latent space
             ngf (int): number of filters in the first layer
             bottom_width (int): size of the first upscaling
             loss_type (str): loss to use in the training ('gan', 'ns', 'hinge',␣
     ↪'wasserstein')
         """
         def __init__(self, nz=16, ngf=16, bottom_width=7, **kwargs):
             super().__init__(nz=nz, ngf=ngf, bottom_width=bottom_width, **kwargs)

             # Build the layers
             self.l1 = nn.Linear(self.nz, (self.bottom_width**2) * self.ngf)
             self.block2 = GBlock(self.ngf, self.ngf, upsample=True)
             self.block3 = GBlock(self.ngf, self.ngf, upsample=True)
             self.b5 = nn.BatchNorm2d(self.ngf)
             self.c5 = nn.Conv2d(self.ngf, 1, 3, 1, padding=1)
             self.activation = nn.ReLU(True)

             # Initialise the weights
             nn.init.xavier_uniform_(self.l1.weight.data, 1.0)
             nn.init.xavier_uniform_(self.c5.weight.data, 1.0)

         def forward(self, x):
             r"""
             Forwards a batch of latent codes z through the generator network.
             end with a sigmoid that gives values [0,1]

             Args:
                 x (Tensor): batch of latent codes, shape (N, nz).

             Returns:
                 Tensor: batch of generated images, shape (N, C, H, W).
             """
             h = self.l1(x)
             h = h.view(x.shape[0], -1, self.bottom_width, self.bottom_width)
             h = self.block2(h)
             h = self.block3(h)
             h = self.b5(h)
             h = self.activation(h)
             h = torch.tanh(self.c5(h))

             return h
```

Now we define the discriminator symmetrically, where an input of (1,28,28) is transformed at the end into logits that will be used to determine if an image is real or obtained from the generator.

[9]:
```
class DCGANDiscriminatorMNIST(dcgan_base.DCGANBaseDiscriminator):
    r"""
    Resnet for the DCGAN discriminator for MNIST, with input (1,28,28) and␣
 ↪output the logit value

    Attributes:
        ndf (int): number of filters in the convolutional layer
    """
    def __init__(self, ndf=16, **kwargs):
        super().__init__(ndf=ndf, **kwargs)

        # Build layers
        self.block1 = DBlockOptimized(1, self.ndf, spectral_norm=False)
        self.block2 = DBlock(self.ndf, self.ndf, downsample=True,␣
 ↪spectral_norm=False)
        self.block3 = DBlock(self.ndf, self.ndf, downsample=False,␣
 ↪spectral_norm=False)
        self.l5 = nn.Linear(self.ndf, 1)
        self.activation = nn.ReLU(True)

        # Initialise the weights
        nn.init.xavier_uniform_(self.l5.weight.data, 1.0)

    def forward(self, x):
        r"""
        Paso forward aplica las diferentes capas

        Args:
            x (Tensor): un batch de imagenes (N, C, H, W).

        Returns:
            Tensor: un batch de logits (N, 1).
        """
        h = x
        h = self.block1(h)
        h = self.block2(h)
        h = self.block3(h)
        h = self.activation(h)
        # Global average pooling
        h = torch.sum(h, dim=(2, 3))
        output = self.l5(h)

        return output
```

As a test we can instantiate the networks and see that they generate tensors of the appropriate size. We should obtain by passing a batch of Gaussian noise vectors a batch of tensors of (batch,1,28,28)

```
[10]: generator = DCGANGeneratorMNIST().to(device)

      fake = generator(torch.rand((8,16)).to(device))
      fake.shape
```

```
[10]: torch.Size([8, 1, 28, 28])
```

We can pass this tensor to the discriminator to see if it generates a vector (batch, 1)

```
[11]: discriminator = DCGANDiscriminatorMNIST().to(device)
      discriminator(fake).shape
```

```
[11]: torch.Size([8, 1])
```

Now we are ready to train the model with MNIST. We will leave the model parameters as default, in this case the default is the `non saturating GAN loss` as the loss function. If you remember, this loss function reversed the direction of how the cross entropy was calculated for the generated samples so that there was a better gradient for them.

We will also define an optimizer for each network, we will assume that the parameters are correct.

```
[12]: generatorDC = DCGANGeneratorMNIST().to(device)
      discriminatorDC = DCGANDiscriminatorMNIST().to(device)

      optD = optim.Adam(discriminatorDC.parameters(), 2e-3, betas=(0.0, 0.9))
      optG = optim.Adam(generatorDC.parameters(), 2e-3, betas=(0.0, 0.9))
```

The training function also allows us to indicate the number of optimizations we make of the discriminator for each epoch, so that it can improve enough to be able to pass information to the generator. We will reduce the learning rate linearly.

```
[13]: trainer = mmc.training.Trainer(netD=discriminatorDC,
                                     netG=generatorDC,
                                     optD=optD,
                                     optG=optG,
                                     n_dis=2,
                                     num_steps=training_steps,
                                     lr_decay='linear',
                                     dataloader=mnist_dl,
                                     log_dir=f'./log/ns_{strftime("%Y%m%d%H%M")}',
                                     print_steps=report_steps,
                                     device=device)
```

```
[14]: trainer.train()
```

Now we can generate digits simply by passing Gaussian noise to the generator

We will first load a generator that has been trained for the longest time

```
[15]: generatorDC.restore_checkpoint('./models/MNIST_DCGAN_netG_10000_steps.pth')
```

```
[15]: 10000
```

```
[16]: nim = 10
      samples = generatorDC.generate_images(nim*nim,device=device)
      fig = plt.figure(figsize=(5,5))
```

```
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow((samples[i].cpu().detach().permute(1,2,0).numpy()+1)/2,␣
 ↪cmap='gray')
    plt.axis('off')
plt.tight_layout(pad=0.)
```

10.6.2 Wasserstein GAN with Gradient Penalty

The Wasserstein GAN changes the training of GANs so that we look for a function that compares the distributions of real and generated data by approximating the Wasserstein distance. We can regularize the optimization by restricting the gradient of interpolated examples between the real data and the generator, keeping it around 1 so that the calculated function is smooth.

Let's define this GAN for the MNIST dataset

[17]:
```
from torch_mimicry.nets.wgan_gp import wgan_gp_base
from torch_mimicry.nets.wgan_gp.wgan_gp_resblocks import DBlockOptimized,␣
 ↪DBlock, GBlock
```

```
class WGANGPGeneratorMNIST(wgan_gp_base.WGANGPBaseGenerator):
    r"""
    Generator ResNet for WGAN-GP.

    Attributes:
        nz (int): dimension of latent noise
        ngf (int): number of filters in the convolutional layer
        bottom_width (int): size of the first upscaling
    """
    def __init__(self, nz=16, ngf=32, bottom_width=7, **kwargs):
        super().__init__(nz=nz, ngf=ngf, bottom_width=bottom_width, **kwargs)

        # Creamos las capas
        self.l1 = nn.Linear(self.nz, (self.bottom_width**2) * self.ngf)
        self.block2 = GBlock(self.ngf, self.ngf, upsample=True)
        self.block3 = GBlock(self.ngf, self.ngf, upsample=True)
        self.b5 = nn.BatchNorm2d(self.ngf)
        self.c5 = nn.Conv2d(self.ngf, 1, 3, 1, padding=1)
        self.activation = nn.ReLU(True)

        # Inicializamos los pesos
        nn.init.xavier_uniform_(self.l1.weight.data, 1.0)

    def forward(self, x):
        r"""
        Forward pass, applies the different layers of the resnet and obtains␣
 ↪an image

        Args:
            x (Tensor): batch of latent codes, shape (N, nz).

        Returns:
            Tensor: batch of generated images, shape (N, C, H, W).
        """
        h = self.l1(x)
        h = h.view(x.shape[0], -1, self.bottom_width, self.bottom_width)
        h = self.block2(h)
        h = self.block3(h)
        h = self.b5(h)
        h = self.activation(h)
        h = torch.tanh(self.c5(h))

        return h
```

[18]:
```
from torch import autograd

class WGANGPDiscriminatorMNIST(wgan_gp_base.WGANGPBaseDiscriminator):
    r"""
    ResNet discriminator para WGAN-GP. Identical to the one from DCGAN but␣
 ↪using
    mean pooling in the output layer.
    Attributes:
        ndf (int): number of filters in the convolutional layer
        gp_scale (float): lambda parameter of the Gradient penalty.
    """
    def __init__(self, ndf=16, **kwargs):
        super().__init__(ndf=ndf, **kwargs)

        # layers
        self.block1 = DBlockOptimized(1, self.ndf)
        self.block2 = DBlock(self.ndf, self.ndf, downsample=True)
        self.block3 = DBlock(self.ndf, self.ndf, downsample=False)
        self.l5 = nn.Linear(self.ndf, 1)

        self.activation = nn.ReLU(True)

        # Weight initialization
        nn.init.xavier_uniform_(self.l5.weight.data, 1.0)

    def forward(self, x):
        r"""
        Forward pass, applies the different layers of the resnet and obtains␣
↪a logit

        Args:
            x (Tensor): batch of images, shape (N, C, H, W).

        Returns:
            Tensor: batch of logits, shape (N, 1).
        """
        h = x
        h = self.block1(h)
        h = self.block2(h)
        h = self.block3(h)
        h = self.activation(h)

        # Global average pooling
        h = torch.mean(h, dim=(2, 3))  # WGAN uses mean pooling instead of␣
↪sum
        output = self.l5(h)

        return output

    def compute_gradient_penalty_loss(self,
                                      real_images,
                                      fake_images,
                                      gp_scale=10.0):
        r"""
        Compute the gradient penalty loss for WGAN-GP.

        Args:
            real_images (Tensor): batch of real images (N, 1, H, W).
            fake_images (Tensor): batch of generated images (N, 1, H, W).
            gp_scale (float): lambda parameter of the Gradient penalty.

        Returns:
            Tensor: scalar tensor of gradient penalty loss.
        """
        # Image parameters
        N, _, H, W = real_images.shape
        device = real_images.device

        # Sample alpha between 0 and 1 for interpolation
        # where alpha is of the same shape to multiply element by element
        alpha = torch.rand(N, 1)
        alpha = alpha.expand(N, int(real_images.nelement() / N)).contiguous()
        alpha = alpha.view(N, 1, H, W)
        alpha = alpha.to(device)

        # Obtains interpolation between real and fake images
        interpolates = alpha * real_images.detach() \
            + ((1 - alpha) * fake_images.detach())
        interpolates = interpolates.to(device)
        interpolates.requires_grad_(True)

        # Obtains the gradients of the interpolates
        disc_interpolates = self.forward(interpolates)
        gradients = autograd.grad(outputs=disc_interpolates,
                                  inputs=interpolates,
                                  grad_outputs=torch.ones(
                                      disc_interpolates.size()).to(device),
                                  create_graph=True,
                                  retain_graph=True,
                                  only_inputs=True)[0]
        gradients = gradients.view(gradients.size(0), -1)

        # Computes the GP
        gradient_penalty = (
            (gradients.norm(2, dim=1) - 1)**2).mean() * gp_scale

        return gradient_penalty
```

[19]:
```
generator = WGANGPGeneratorMNIST().to(device)

fake = generator(torch.rand((8,16)).to(device))
fake.shape
```

[19]:
```
torch.Size([8, 1, 28, 28])
```

[20]:
```
discriminator = WGANGPDiscriminatorMNIST().to(device)
discriminator(fake).shape
```

[20]:
```
torch.Size([8, 1])
```

[21]:
```
generatorW = WGANGPGeneratorMNIST().to(device)
discriminatorW = WGANGPDiscriminatorMNIST().to(device)

optD = optim.Adam(discriminatorW.parameters(), 2e-3, betas=(0.0, 0.9))
optG = optim.Adam(generatorW.parameters(), 2e-3, betas=(0.0, 0.9))
```

The training function also allows us to indicate the number of optimizations we make of the discriminator for each epoch, so that it can improve enough to be able to pass information to the generator. We will reduce the learning rate linearly.

[22]:
```
trainer = mmc.training.Trainer(netD=discriminatorW,
                                netG=generatorW,
                                optD=optD,
                                optG=optG,
                                n_dis=2,
                                num_steps=training_steps,
                                lr_decay='linear',
                                dataloader=mnist_dl,
                                log_dir=f'./log/
 ↪wass_{strftime("%Y%m%d%H%M")}',
                                print_steps=report_steps,
                                device=device)
```

[23]:
```
trainer.train()
```

[24]:
```
nim = 10
samples = generatorW.generate_images(nim*nim,device=device)
fig = plt.figure(figsize=(5,5))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow((samples[i].cpu().detach().permute(1,2,0).numpy()+1)/2,␣
 ↪cmap='gray')
    plt.axis('off')
plt.tight_layout(pad=0.)
```

10.6.3 Latents interpolation

Just as we can do with other models, we can navigate the latent space by interpolating between them.

In a GAN, what we expect is that when interpolating between latents, the transition between examples is minimally smooth, indicating that there are no abrupt transitions between the different modes that have been learned.

We will load a generator that has been trained for a longer time.

[25]:
```
generatorW.restore_checkpoint('/content/models/MNIST_WGANGP_netG_10000_steps.
  ↪pth')
```

[25]:
```
10000
```

The problem we have with GANs that are not conditioned is that we do not know what we will obtain from a random latent, so we will generate two latents assuming that they correspond to different digits (otherwise we can simply run the cell several times until it is the case ;-)

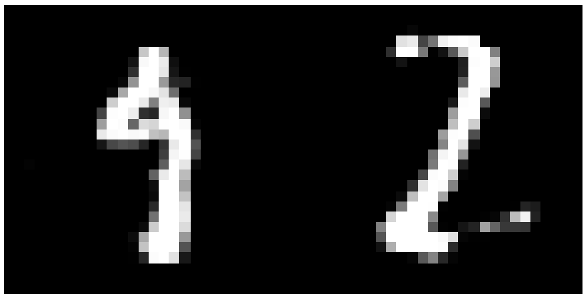
[26]:
```
lat1 = torch.randn(1,16)
lat2 = torch.randn(1,16)
```

Now we pass the latents through the generator to obtain the samples

[27]:
```
img1 = generatorW(lat1.unsqueeze(dim=1).to(device))
img2 = generatorW(lat2.unsqueeze(dim=1).to(device))
```

We hope they are good enough ;-)

[28]:
```
fig = plt.figure(figsize=(4,4))
plt.subplot(1, 2, 1)
plt.imshow((img1.squeeze().cpu().detach().numpy()+1)/2, cmap='gray');
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow((img2.squeeze().cpu().detach().numpy()+1)/2, cmap='gray');
plt.axis('off')
plt.tight_layout(pad=0.)
```

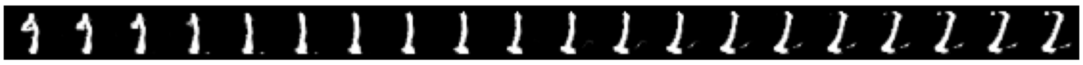


Now we can simply generate latents from the vector that joins the two generated latents

[29]:
```
interp = []

for v in torch.arange(0,1,0.05):
    interp.append(torch.lerp(lat1,lat2, v))
```

and we can see how smooth the transition is

[30]:
```
fig = plt.figure(figsize=(15,10))
for i, v in enumerate(interp):
    plt.subplot(1, len(interp), i+1)
    plt.imshow((generatorW(v.to(device)).squeeze().cpu().detach().numpy()+1)/
 ↪2, cmap='gray')
    plt.axis('off')
plt.tight_layout(pad=0.)
```



10.6.4 Conditional GAN

We are now going to use a GAN in which we can control what is generated. Specifically, we will use cGAN with projection in the discriminator. This introduces the class that we want to obtain in the generator and modifies the discriminator so that apart from obtaining the logits we add the result of projecting them on an embedding of the classes

[31]:
```python
from torch_mimicry.nets.cgan_pd import cgan_pd_base
from torch_mimicry.modules import SNLinear, SNEmbedding

class CGANPDGeneratorMNIST(cgan_pd_base.CGANPDBaseGenerator):
    r"""
    Generator ResNet for cGAN-PD,

    Attributes:
        num_classes (int): Number of classes to condition on
        bottom_width (int): size of the first upscaling
        nz (int): Dimension of the latent space
        ngf (int): Number of channels in the convolutions
    """
    def __init__(self, num_classes, bottom_width=7, nz=16, ngf=16, **kwargs):
        super().__init__(nz=nz,
                         ngf=ngf,
                         bottom_width=bottom_width,
                         num_classes=num_classes,
                         **kwargs)

        # layers
        self.l1 = nn.Linear(self.nz, (self.bottom_width**2) * self.ngf)
        self.block2 = GBlock(self.ngf,
                             self.ngf,
                             upsample=True,
                             num_classes=self.num_classes)
        self.block3 = GBlock(self.ngf,
                             self.ngf,
                             upsample=True,
                             num_classes=self.num_classes)
        self.b5 = nn.BatchNorm2d(self.ngf)
        self.c5 = nn.Conv2d(self.ngf, 1, 3, 1, padding=1)
        self.activation = nn.ReLU(True)

        # weight initialization
        nn.init.xavier_uniform_(self.l1.weight.data, 1.0)
        nn.init.xavier_uniform_(self.c5.weight.data, 1.0)

    def forward(self, x, y=None):
        r"""
        Forward pass. Applies the different layers of the resnet and obtains␣
 ↪an image
        conditioned on the class

        Args:
            x (Tensor): batch of latent codes, shape (N, nz).
            y (Tensor): batch of labels, shape (N, num_classes).

        Returns:
            Tensor: batch of generated images, shape (N, C, H, W).
        """
        if y is None:
            y = torch.randint(low=0,
                              high=self.num_classes,
                              size=(x.shape[0], ),
                              device=x.device)

        h = self.l1(x)
        h = h.view(x.shape[0], -1, self.bottom_width, self.bottom_width)
        h = self.block2(h, y)
        h = self.block3(h, y)
        h = self.b5(h)
        h = self.activation(h)
        h = torch.tanh(self.c5(h))

        return h
```

[32]:
```
class CGANPDDiscriminatorMNIST(cgan_pd_base.CGANPDBaseDiscriminator):
    r"""
    ResNet Discriminator for cGAN-PD.

    Attributes:
        num_classes (int): Number of classes to condition on
        ndf (int): Number of channels in the convolutions
    """
    def __init__(self, num_classes, ndf=16, **kwargs):
        super().__init__(ndf=ndf, num_classes=num_classes, **kwargs)

        # Capas
        self.block1 = DBlockOptimized(1, self.ndf)
        self.block2 = DBlock(self.ndf, self.ndf, downsample=True)
        self.block3 = DBlock(self.ndf, self.ndf, downsample=False)
        self.l5 = SNLinear(self.ndf, 1)

        # Embedding para la clase
        self.l_y = SNEmbedding(num_embeddings=self.num_classes,
                               embedding_dim=self.ndf)

        # Inicializamos los pesos
        nn.init.xavier_uniform_(self.l5.weight.data, 1.0)
        nn.init.xavier_uniform_(self.l_y.weight.data, 1.0)

        self.activation = nn.ReLU(True)

    def forward(self, x, y=None):
        r"""
        Forward pass. Applies the different layers of the resnet and obtains␣
↪a logit
        Performs a forward with a projection of the class to condition the␣
↪logits

        Args:
            x (Tensor): batch of latent codes, shape (N, nz).
            y (Tensor): batch of labels, shape (N, num_classes).

        Returns:
            Tensor: batch of generated images, shape (N, C, H, W).
        """
        h = x
        h = self.block1(h)
        h = self.block2(h)
        h = self.block3(h)
        h = self.activation(h)

        # Global sum pooling
        h = torch.sum(h, dim=(2, 3))
        output = self.l5(h)

        # Projection of the class summed to the logits
        w_y = self.l_y(y)
        output += torch.sum((w_y * h), dim=1, keepdim=True)

        return output
```

[33]:
```
generator = CGANPDGeneratorMNIST(num_classes=10).to(device)

fake = generator(torch.rand((8,16)).to(device), torch.tensor([1]*8).
 ↪to(device))
fake.shape
```

[33]:
```
torch.Size([8, 1, 28, 28])
```

[34]:
```
discriminator = CGANPDDiscriminatorMNIST(num_classes=10).to(device)
discriminator(fake, torch.tensor([1]*8).to(device)).shape
```

[34]:
```
torch.Size([8, 1])
```

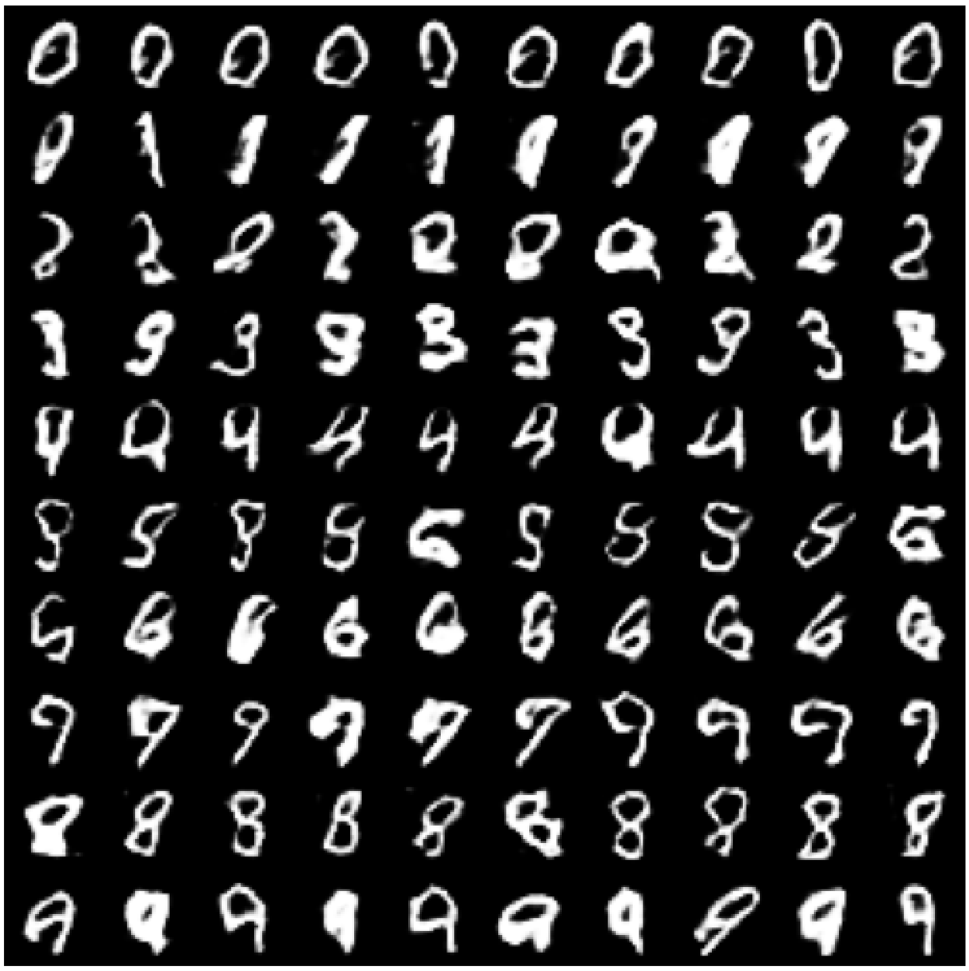
[35]:
```
generatorC = CGANPDGeneratorMNIST(num_classes=10).to(device)
discriminatorC = CGANPDDiscriminatorMNIST(num_classes=10).to(device)

optD = optim.Adam(discriminatorC.parameters(), 2e-3, betas=(0.0, 0.9))
optG = optim.Adam(generatorC.parameters(), 2e-3, betas=(0.0, 0.9))
```

The training function also allows us to indicate the number of optimizations we make of the discriminator for each epoch, so that it can improve enough to be able to pass information to the generator. We will reduce the learning rate linearly.

[36]:
```
trainer = mmc.training.Trainer(netD=discriminatorC,
                                netG=generatorC,
                                optD=optD,
                                optG=optG,
                                n_dis=2,
                                num_steps=training_steps,
                                lr_decay='linear',
                                dataloader=mnist_dl,
                                        log_dir=f'./log/
↪cgan_{strftime("%Y%m%d%H%M")}',
                                        print_steps=report_steps,
                                        device=device)
```

[37]:
```
trainer.train()
```

[38]:
```
nim = 10

fig = plt.figure(figsize=(5,5))
for j in range(10):
    for i in range(nim):
        samples = generatorC.generate_images(nim,c=torch.tensor(j).
↪to(device),device=device)
        plt.subplot(nim, 10, (j*10)+i+1)
        plt.imshow((samples[i].cpu().detach().permute(1,2,0).numpy()+1)/2,␣
↪cmap='gray')
        plt.axis('off')
plt.tight_layout(pad=0.)
```



10.6.5 Interpolation with control

The fact that we can control the class that is generated can be used to generate variations between examples of the same class. In this case the GAN should have separated the classes from each other.

We will use the trained model for a longer time, you will find it in with the acompanying material, you just have to put it in the /content/models directory

```
[39]: generatorC.restore_checkpoint('/content/models/MNIST_cGAN_netG_10000_steps.
        ↪pth')
```

```
[39]: 10000
```

As before, we generate two random latents and set the class we want to obtain

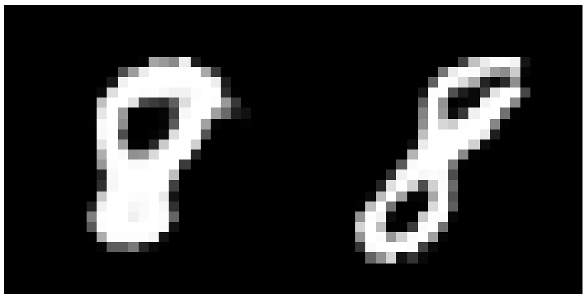
```
[40]: lat1 =  torch.randn(1,16)
      lat2 = torch.randn(1,16)
      cl = 8
```

Now we pass the latents through the generator to obtain the samples

```
[41]: img1 = generatorC(lat1.unsqueeze(dim=1).to(device), torch.tensor([cl]).
        ↪to(device))
      img2 = generatorC(lat2.unsqueeze(dim=1).to(device), torch.tensor([cl]).
        ↪to(device))
```

We can try to find two examples that are different enough for a change to be seen when interpolating

```
[42]: fig = plt.figure(figsize=(4,4))
      plt.subplot(1, 2, 1)
      plt.imshow((img1.squeeze().cpu().detach().numpy()+1)/2, cmap='gray');
      plt.axis('off')
      plt.subplot(1, 2, 2)
      plt.imshow((img2.squeeze().cpu().detach().numpy()+1)/2, cmap='gray');
      plt.axis('off')
      plt.tight_layout(pad=0.)
```



Now we can simply generate latents from the vector that joins the two generated latents

```
[43]: interp = []

      for v in torch.arange(0,1,0.05):
          interp.append(torch.lerp(lat1,lat2, v))
```

and we can see how smooth the transition is

[44]:
```
fig = plt.figure(figsize=(15,10))
for i, v in enumerate(interp):
    plt.subplot(1, len(interp), i+1)
    plt.imshow((generatorC(v.to(device), torch.tensor([cl]).to(device)).
 ↪squeeze().cpu().detach().numpy()+1)/2, cmap='gray')
    plt.axis('off')
plt.tight_layout(pad=0.)
```

10.6.6 StyleGAN2

StyleGAN is a set of models that enable high-quality image generation in specific domains.

We will use pre-trained StyleGAN2 models to test its performance by generating different images and manipulating their latent space.

As you will remember, this network is different from the usual GANs. It has a network (MLP) that maps a latent Gaussian to a new space that determines the style of the image (style code). This vector is used as input for the different scaling layers of the network that generates the image. The generator uses this style vector to manipulate the input of each layer determining different things depending on the scale.

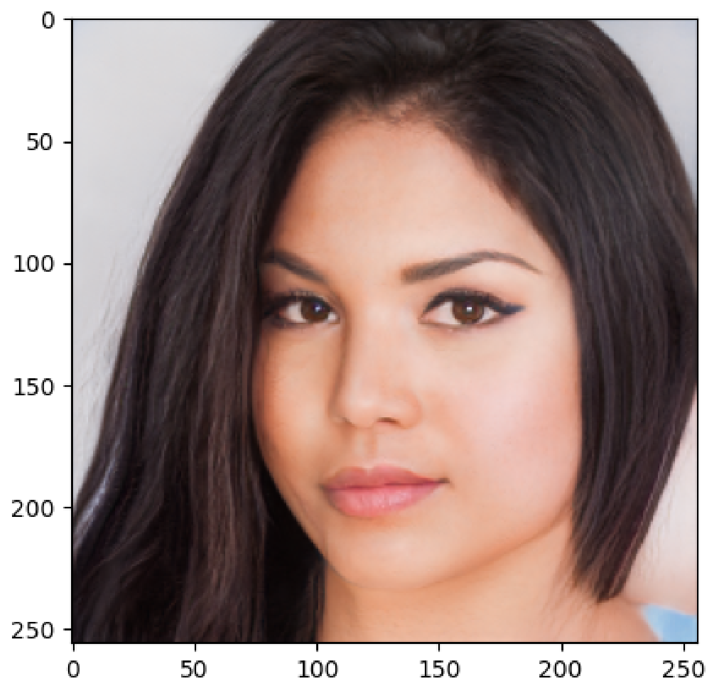
If we load the pretrained model.

[49]:
```
with open('/content/stylegan2-ffhq-256x256.pkl', 'rb') as f:
    G = pickle.load(f)['G_ema'].cuda()  # torch.nn.Module
```

In G we have the complete network and we can generate an image by simply generating a random latent and making a step forward of the network.

[50]:
```
# We define a random number generator and set an initial seed so that
# we all get the same thing, but it can changed to whatever we want
gen = torch.Generator()
gen.manual_seed(12345678);
```

[51]:
```
# Latent
z = torch.randn([1, G.z_dim], generator=gen).cuda()
# Some models allow conditioning by class, the one we will use is␣
 ↪unconditioned
c = None
img = G(z, c)
plt.imshow(torch.clamp((img.cpu().squeeze().detach().permute(1,2,0)+1)/2,␣
 ↪min=0, max=1).numpy());
```



Interpolation

As in the previous networks we can also do interpolation between latents. We can generate two random images and obtain a set of images that transition between the two.

We set again the seeds that generate the noise for reproducibility, but you can change them to any value

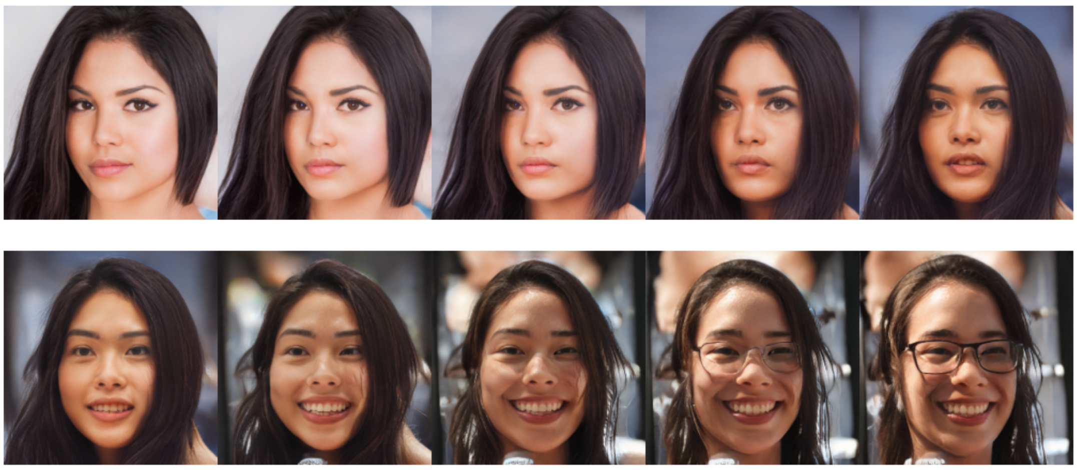
[52]:
```
gen.manual_seed(12345678);
lat1 = torch.randn([1, G.z_dim], generator=gen).cuda()
gen.manual_seed(9876543);
lat2 = torch.randn([1, G.z_dim], generator=gen).cuda()
interp = []

for v in torch.arange(0,1,0.1):
    interp.append(torch.lerp(lat1,lat2, v.to(device)))
```

[53]:
```
img1 = G(lat1, c)
img2 = G(lat2, c)
fig = plt.figure(figsize=(6,3))
plt.subplot(1, 2, 1)
plt.imshow(torch.clamp((img1.cpu().squeeze().detach().permute(1,2,0)+1)/2,␣
 ↪min=0, max=1).numpy());
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(torch.clamp((img2.cpu().squeeze().detach().permute(1,2,0)+1)/2,␣
 ↪min=0, max=1).numpy());
plt.axis('off')
plt.tight_layout(pad=0.)
```

You will see that the transitions are quite smooth, even when the data is now more complex, transforming the different elements contained in the image. StyleGAN2's ability to disentangle meaningful attributes in the space generated by the transformation network makes it easier for this to work well (most of the time :-)

In the example with the seeds you can see how each transition changes the angle of the face, the hairstyle, the eyes, the smile and glasses appearing...

[54]:
```
fig = plt.figure(figsize=(10,5))
for i, v in enumerate(interp):
    plt.subplot(2, len(interp)//2, i+1)
    img = G(v, c)
    plt.imshow(torch.clamp((img.cpu().squeeze().detach().permute(1,2,0)+1)/
 ↪2, min=0, max=1).numpy());
    plt.axis('off')
plt.tight_layout(pad=0.)
```



Manipulation of the style codes

The style codes generated by the transformation network from the latent Gaussian control different elements of the content depending on the entry point in the generation network.

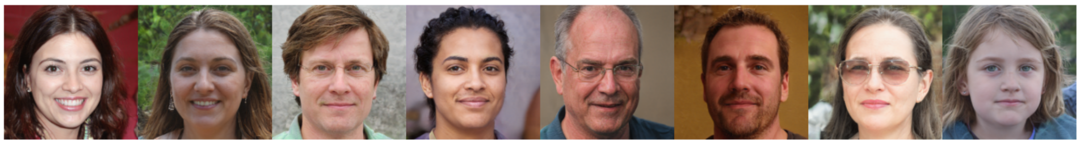
In the generation network, the lowest resolutions (4-8) control the content by changing fundamental elements, the intermediate ones (16-32) control finer features of the face, for example the eyes, the highest ones (64-256) the color scheme and textures.

We will select a group of "random" images to do experiments with the style. The style code of the first half of the images will be changed to that of the other half in the different resolutions.

[55]:
```
seeds = [3233338, 1254678, 2387498, 98329239, 1112279,230,50,155465]
images = []
styles = []

for i in range(len(seeds)):
    gen.manual_seed(seeds[i]);
    z = torch.randn([1, G.z_dim], generator=gen).cuda()
    w = G.mapping(z, c, truncation_psi=0.5, truncation_cutoff=8)
    styles.append(w)
    images.append(G.synthesis(w, noise_mode='const', force_fp32=True))
```

A certain variety of images has been chosen based on age, sex and other additional elements

[56]:
```
fig = plt.figure(figsize=(12,3))
for i in range(len(seeds)):
    plt.subplot(1, len(seeds), i+1)
    plt.imshow(torch.clamp((images[i].cpu().squeeze().detach().
 ↪permute(1,2,0)+1)/2, min=0, max=1).numpy());
    plt.axis('off')
plt.tight_layout(pad=0.)
```

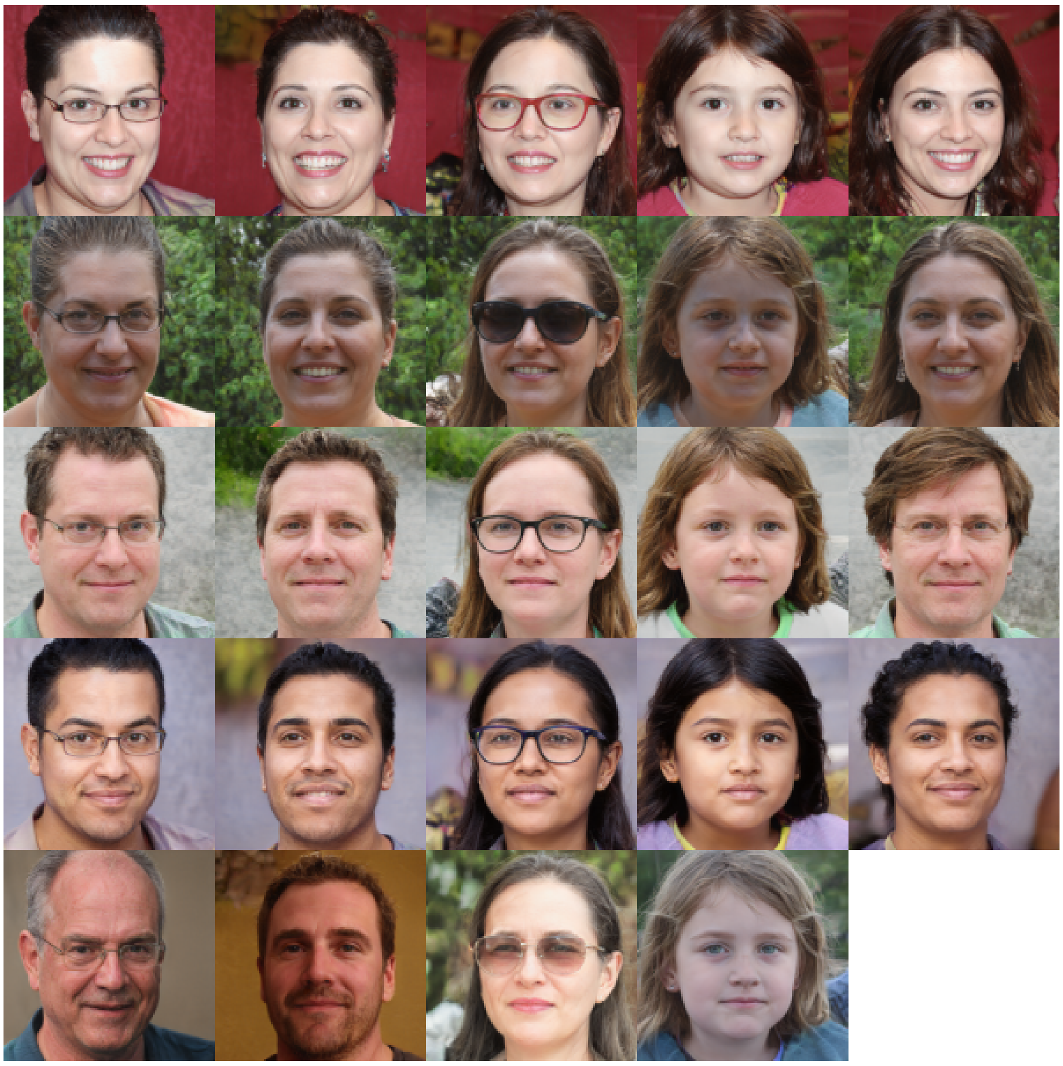


This function generates the style code combinations for a resolution

[57]:
```
layers={'coarse':[0,4], 'middle':[4,8], 'fine':[8,14]}

def mix_style(mode):
    fig = plt.figure(figsize=(6,6))
    for i in range(len(styles)//2):
        for j in range(len(styles)//2):
            plt.subplot(len(styles)//2+1, len(styles)//2+1, (i*(len(styles)//
 ↪2))+ j+i+1)
            w = styles[i].clone()
            for l in range(layers[mode][0],layers[mode][1]):
                w[0,l]= styles[j+(len(styles)//2)][0,l]
            img = G.synthesis(w, noise_mode='const', force_fp32=True)
            plt.imshow(torch.clamp((img.cpu().squeeze().detach().
 ↪permute(1,2,0)+1)/2, min=0, max=1).numpy());
            plt.axis('off')
        plt.subplot(len(styles)//2+1, len(styles)//2+1, (i*(len(styles)//
↪2))+ j +i +2)
        plt.imshow(torch.clamp((images[i].cpu().squeeze().detach().
↪permute(1,2,0)+1)/2, min=0, max=1).numpy());
        plt.axis('off')

    for i in range(len(styles)//2):
        plt.subplot(len(styles)//2+1, len(styles)//2+1, (len(styles)//2) *␣
↪(len(styles)//2 +1) +i +1)
        plt.imshow(torch.clamp((images[i+(len(styles)//2)].cpu().squeeze().
↪detach().permute(1,2,0)+1)/2, min=0, max=1).numpy());
        plt.axis('off')
    plt.tight_layout(pad=0.)
```

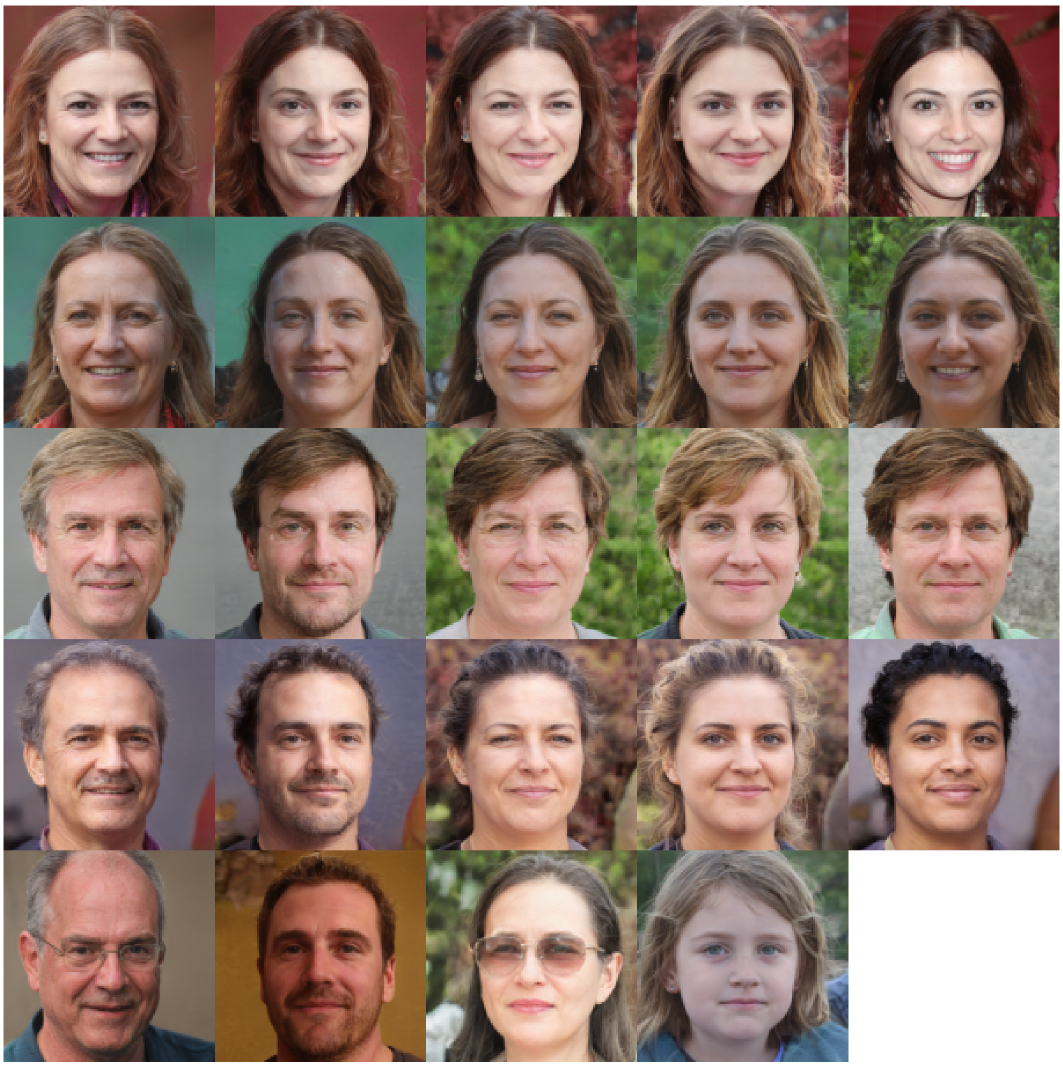
We see that the low resolution mix changes the content of the original images quite a bit, respecting certain elements, such as the background, but changing, for example, the position of the head, age or sex and adding additional elements, such as glasses

[58]:
```
mix_style('coarse')
```



Intermediate resolutions change finer details, such as beard (where it makes sense) or hair color

```
[59]: mix_style('middle')
```



Higher resolutions change the color scheme

```
[60]: mix_style('fine')
```

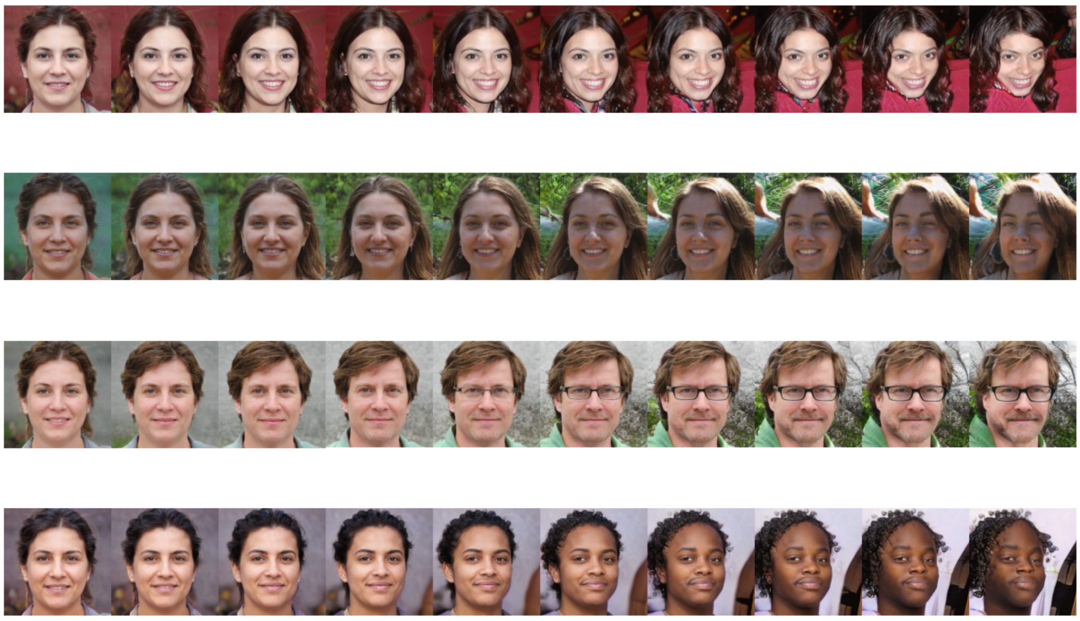
10.6.7 The truncation trick

If you remember, BigGAN introduced a *trick* to improve the general quality of the samples. This consisted of limiting the maximum value that we admit in the latent Gaussian, changing the values that go beyond that value. This trick is already used in a general way, for example in StyleGAN.

A problem that it generates is the variety of the samples, the more we limit it, the less diversity our data set will have. We can see this below. We have generated all the samples with a truncation value of 0.5.

Exploring the range from 0 to 1.5, we can see that with the value 0 (first column) all the faces are the same and as we relax it, distorted images begin to appear.

[61]:
```
fig = plt.figure(figsize=(12,15))
vals = np.arange(0,1.5, 0.15)
pos =0
for i in range(len(seeds)//2):
    for tr in vals:
        pos+=1
        gen.manual_seed(seeds[i]);
        z = torch.randn([1, G.z_dim], generator=gen).cuda()
        w = G.mapping(z, c, truncation_psi=tr, truncation_cutoff=8)
        img = G.synthesis(w, noise_mode='const', force_fp32=True)
        plt.subplot(len(seeds), len(vals), pos)
        plt.imshow(torch.clamp((img.cpu().squeeze().detach().
↪permute(1,2,0)+1)/2, min=0, max=1).numpy());
        plt.axis('off')
plt.tight_layout(pad=0.)
```



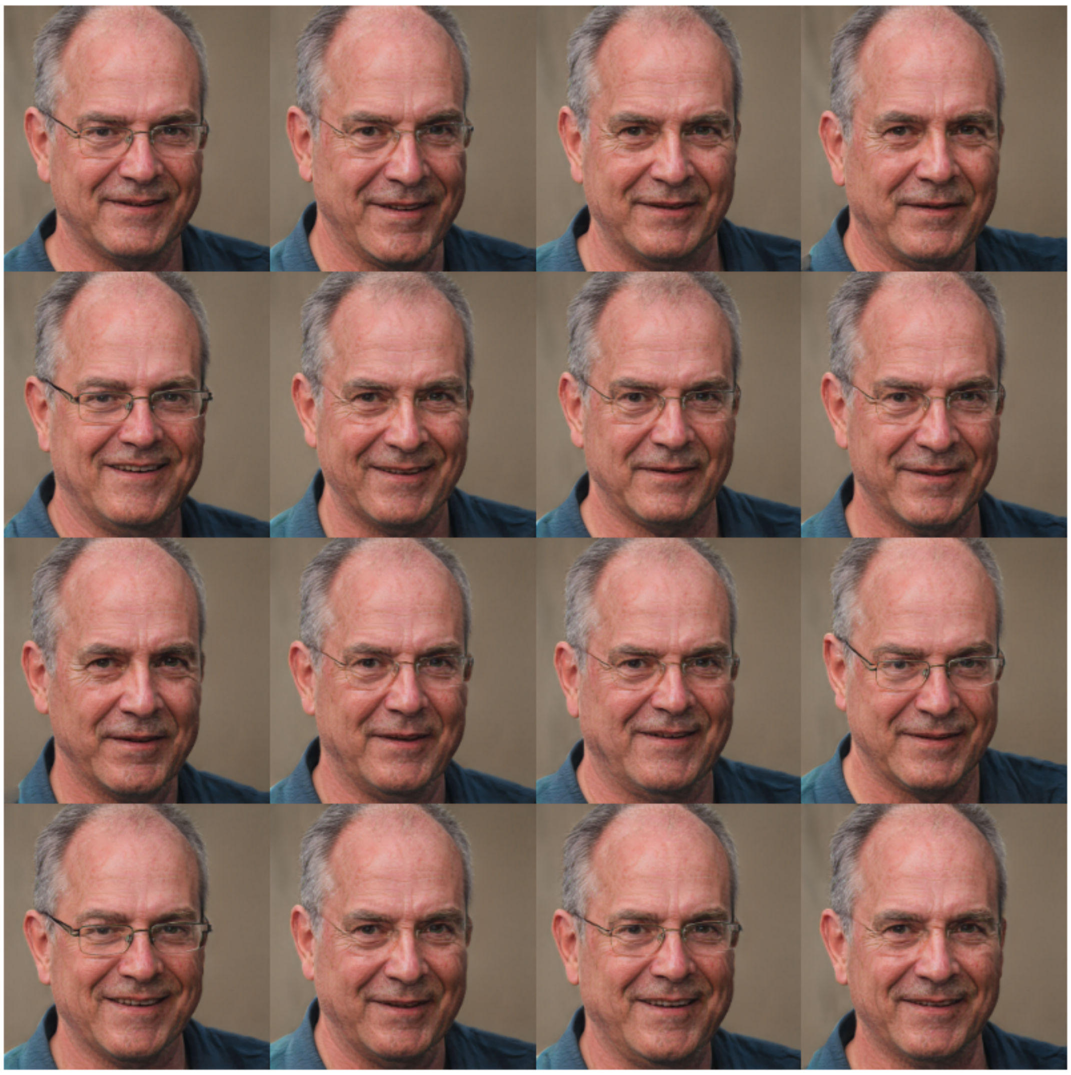
10.6.8 Consistency on the sample space

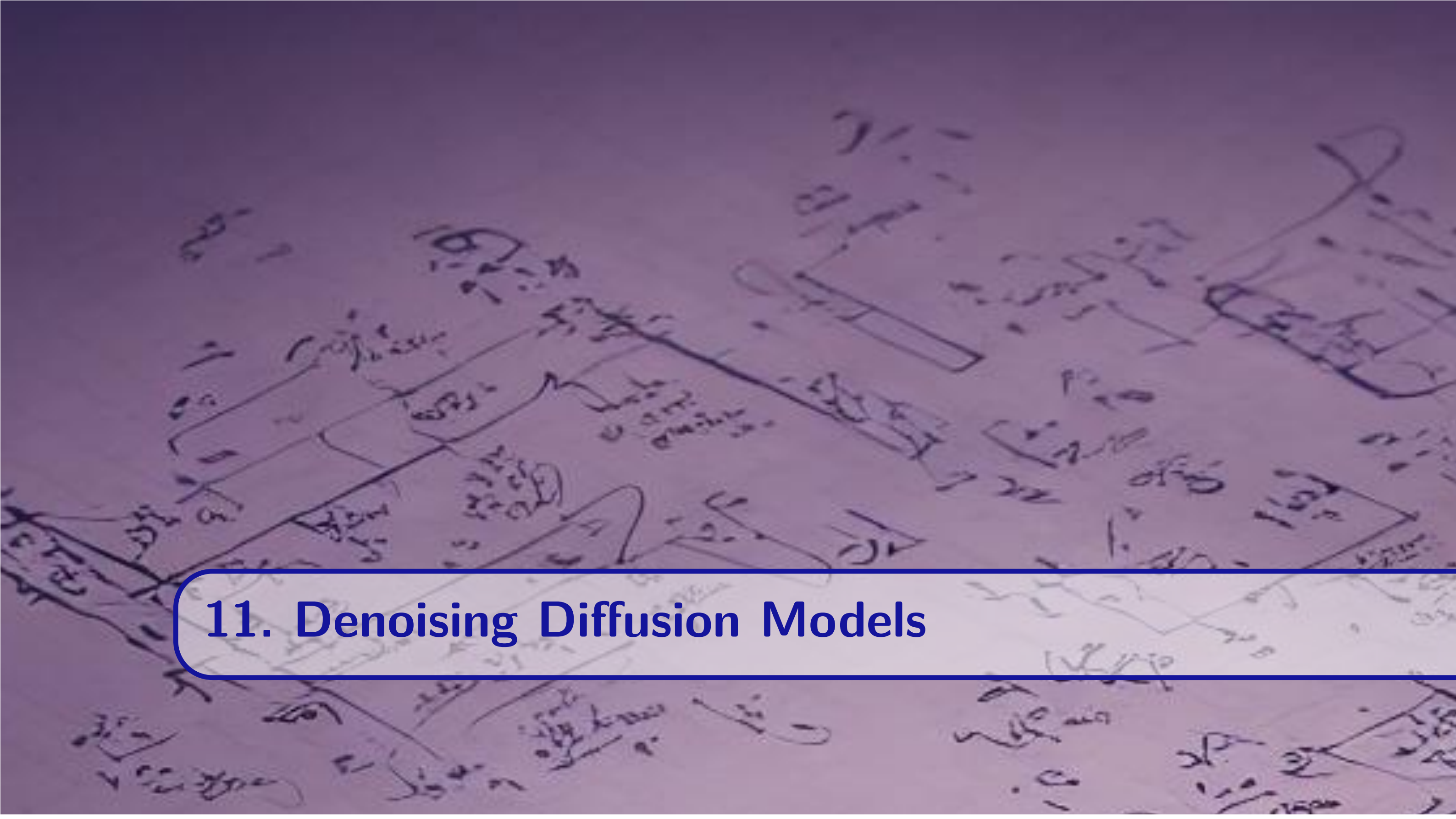
As a final experiment, one thing we can assume from the space generated by a GAN around the same latent is that the samples will be similar. In StyleGAN this also happens within the style code space. For example, if we add a slight Gaussian noise to the style code that controls the content of the generation the samples should be similar. Style vectors have a size of 512.

You can try what happens as the noise increases, or what happens if the noise that we add to each of the 4 codes is different or if we change the style codes of other resolutions.

[62]:
```
# Generate a style code from one of the previous seeds
gen.manual_seed(seeds[4]);
z = torch.randn([1, G.z_dim], generator=gen).cuda()
w = G.mapping(z, c, truncation_psi=0.5, truncation_cutoff=8)

# Scaling of the gaussian noise
delta = torch.tensor([0.2]).to(device)

nim =4
fig = plt.figure(figsize=(8,8))
```

```
for i in range(nim*nim):
    pos+=1
    vec = w.clone()
    vec[0,0:4] =  vec[0,0:4]+ (torch.randn([512]).cuda()  * delta )
    img = G.synthesis(vec , noise_mode='const', force_fp32=True)
    plt.subplot(nim, nim, i+1)
    plt.imshow(torch.clamp((img.cpu().squeeze().detach().permute(1,2,0)+1)/
↪2, min=0, max=1).numpy());
    plt.axis('off')
plt.tight_layout(pad=0.)
```





11. Denoising Diffusion Models

11.1 Diffusion

Diffusion refers, in general, to a physical process where something goes from a state of concentration to a more spread state. A visual example is when a drop of dye is put in water, and it slowly expands until all the liquid is of a uniform color. We can describe it as moving from a state of order to a state of chaos/disorder. This process behaves following a function that depends on time, that is the diffusion process.

In the physical world this process is in general irreversible, but mathematically we can see it as two-way process. We can use this property for defining models that are able to obtain samples directly from random noise. This is the fundamental idea of denoising diffusion models, to use this reversibility for generating data from noise (chaos).

11.2 Denoising Diffusion Models (DDMs)

The generation process of DDMs is similar to GANs or Flows, it starts with a gaussian noise latent. The main difference is that the latent has to be transformed using a diffusion process. The training of the models is performed using data obtained from samples which content has been gradually destroyed using gaussian noise with increasing intensity that ends as just gaussian noise. Later we will see that the diffusion process can be obtained using other distributions. That will be necessary when we can not assume that our samples are composed by continuous variables.

The diffusion process has to hold some constraints:

- For each step, the process must follow a simple distribution (e.g. independent gaussian) so it is possible to estimate the forward and backward process
- The difference between consecutive steps must be small enough, so it also is easy to estimate, and follows a simple distribution (e.g. gaussian)
- The forward process is deterministic, and it is cheap to compute, so we can obtain noisy

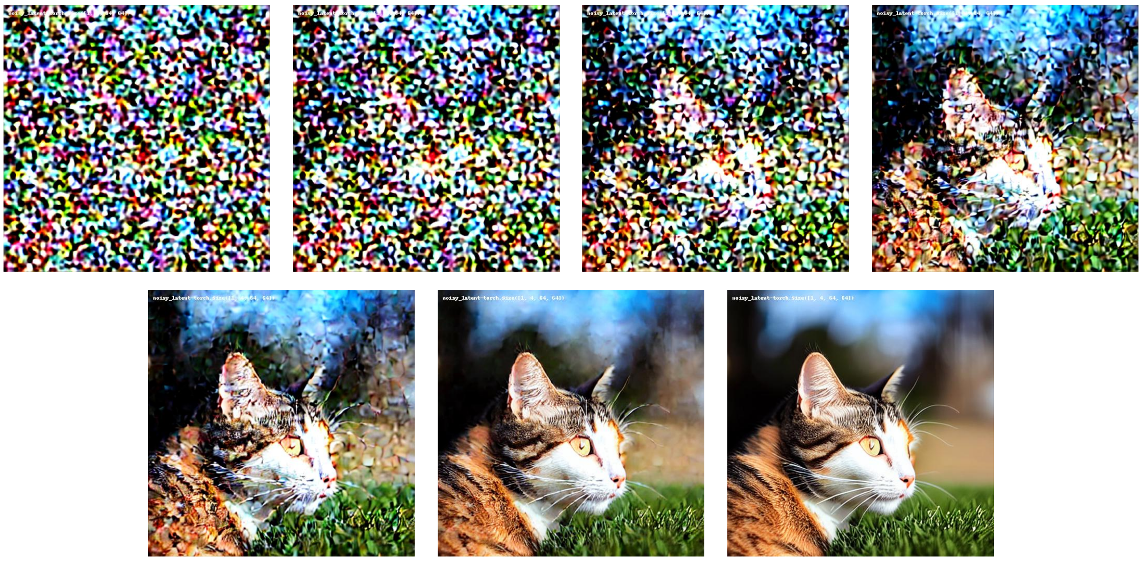


Figure 11.1: Reverse diffusion of an image

samples at will at any point of the diffusion process

This process can be performed with different kinds of samples, but most of the applications that we will explain will be about images. Figure 11.1 shows the reverse process that is performed by a DDM. Each step some noise is taken from the previous step following a path that ends on a real image.

There are three main families of generative diffusion models that are related to each other and have appeared in succession. The first one is **Denoising Diffusion Probabilistic Models** (DDPMs), where the forward process is defined as a discrete markov process, each step is a probabilistic transition that transforms the data from the original sample to gaussian noise, and the reverse process travels the markov chain backwards. The second one is **Noise Conditioned Score Networks** (NSCN), the forward process is also a markov chain and a network tries to match noised samples to a distribution, the fitting uses *score matching* to explore the space of distributions. The third one includes the other two as particular cases, **Score Based Generative Modeling with Stochastic Differential Equations** defines the forward process as a continuous process in time and can be defined as a differential equation's system, that is solved in reverse time to obtain samples.

11.3 Denoising Diffusion Probabilistic Models (DDPMs)

DDPMs [66] are latent variable models that define a model with T steps. This means that the sample has to be transformed several times to obtain the final result. This makes it different from previous models where a sample is obtained in just one step, a forward pass using a neural network. These steps connect the real samples (x_0) with samples from a gaussian distribution (x_T). As in flows, during all the steps the latents have the dimensionality of the data, the difference is than in the networks implementing flows the dimensionality is maintained for each layer and the samples are obtained in one pass, for DDPMs the neural network is computed several times that will have the same dimensionality only at the input and the output and will pass through a latent space like VAEs. The joint probability distribution of all the steps $p_\theta(x_{0:T})$

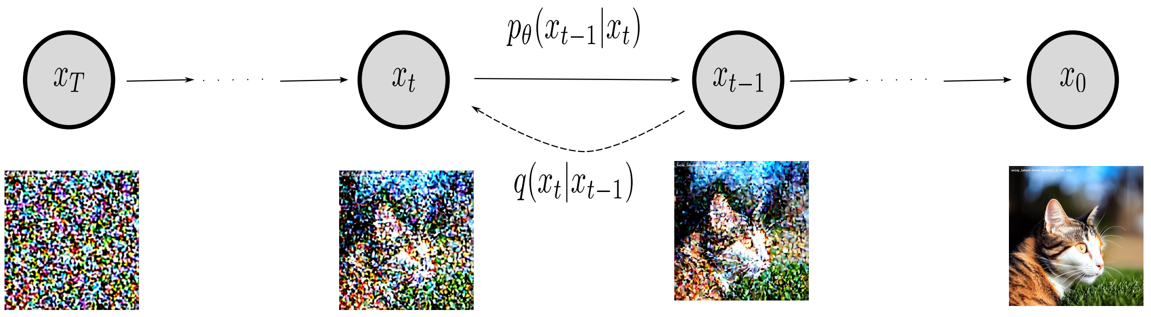


Figure 11.2: Forward and reverse process of DDPMs

is called the reverse process, where:

$$p_\theta(x_0) = \int p_\theta(x_{0:T})dx_{1:T}$$

To make things tractable the gaussian distribution is used for all the elements of the model. The joint probability distribution of the generative process can be defined as a markov chain with transitions defined by gaussian distributions. The initial state is defined by the gaussian noise with unit diagonal variance and zero mean:

$$p(X_T) = \mathcal{N}(x_T; 0, I)$$

When the number of steps in the markov process is large, the difference between two consecutive steps is very small, this allows defining the transitions as gaussians that depend on the step t of the markov process as:

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t))$$

Defining this distribution as autoregressive, this makes the joint distribution:

$$p_\theta(x_{0:T}) = p(x_T)\prod_{t=1}^{t} p_\theta(x_{t-1}|x_t))$$

Now the goal is to learn the parameters of these gaussian distributions. Figure 11.2 shows the forward and reverse process of DDPMs. The distribution p will be our generative process, we will reverse the noising of the sample one step at a time. The forward process is defined by q.

The forward-backward process makes the whole transformation similar to VAEs, where the encoder network transforms the input to a gaussian latent, and the decoder obtains from the gaussian latent new samples. However, the forward process of this model is different from VAEs. The forward step is where the diffusion process is applied, transforming the sample onto gaussian noise. This process is **deterministic** and adds gaussian noise subjected to a variance schedule $(\beta_1, \ldots, \beta_T)$. This means that the variance is increased following a function of t. The forward process is also a markov chain with transition probabilities:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{\beta_t}x_{t-1}, \beta_t I)$$

The joint probability distribution of the forward process can be written as an autoregressive distribution:

$$q(x_{1:T}|x_0) = \prod_{t=1}^{t} q(x_t|x_{t-1})$$

The advantage of this formulation (deterministic) for the forward process is that we can obtain samples at any time step t in closed form. This is important given that we will have to learn from these samples to reverse the diffusion process from two consecutive steps. We can define:

$$\alpha_t = 1 - \beta_t \quad \bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$$

That allows defining the sampling distribution for x_t as:

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t} x_0, (1-\bar{\alpha}_t)I)$$

So we can obtain sample from x_0 for any time step t as:

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{(1-\bar{\alpha}_t)}\epsilon \;\; with \;\; \epsilon \sim \mathcal{N}(0, I)$$

The reverse process starts by sampling x_T from the gaussian distribution as we usually do on other models, but now x_T has the same dimensionality as the data. Then we obtain iteratively x_{t-1} sampling from $q(x_{t-1}|x_t)$ until reaching the number of steps of the diffusion process. This should end on a sample from the data distribution.

The distribution $q(x_{t-1}|x_t) \propto q(x_{t-1})q(x_t|x_{t-1})$ unfortunately is intractable. The good news is that if the variance β_t used at each step is sufficiently small, this can be approximated using a gaussian distribution. This will imply that t has to be a large number (sometimes in the order of thousands). In another section we will talk about different methods about how to reduce this number of steps to a smaller one. Think that each estimation is a forward pass through a neural network and we will have to predict the mean and variance for each variable of the sample.

As in VAEs, when we have a distribution that we can not estimate directly, we can always use a surrogate distribution to approximate the one we can not obtain. We can also use the ELBO for obtaining a loss that optimizes the distance between the surrogate distribution and the one that we can not compute.

We want to obtain a distribution $p_\theta(x_0)$ as close as possible to the real distribution, the ELBO for its log likelihood corresponds to (see paper for derivation):

$$\mathbb{E}_{q(x_0)}[-\log p_\theta(x_0)] \leq \mathbb{E}_{q(x_0)q(x_{1:T}|x_0)}\left[-\log p(x_T) - \sum_{t\geq 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_t|x_{t-1})}\right]$$

The optimization of random terms of this ELBO can be used to learn the distribution using gradient descent. This means to chose for a sample a time step at random, and to generate two noised samples using the forward process (this is deterministic) at steps $t-1$ and t for learning the distribution.

For applying gradient descent, we can use the KL divergence to measure the distance among distributions obtaining the loss:

$$\mathcal{L} = \mathbb{E}_q \left[KL(q(x_T|x_0)||p(x_T)) + \sum_{t>1} KL(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t)) - \log p_\theta(x_0|x_1)\right]$$

The distribution $q(x_{t-1}|x_t, x_0)$ is tractable and can be defined as:

$$q(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}, \tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t I)$$

with

$$\tilde{\mu}_t(x_t, x_0) = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{1-\beta_t}(1-\alpha_{t-1})}{1-\alpha_t} x_t \quad \tilde{\beta}_t = \frac{1-\alpha_{t-1}}{1-\alpha_t}\beta_t$$

Respect to the parameters of the p distributions, there are different ways of choosing their variance. In the original DDPM paper this is fixed to $\sigma^2{}_t = \beta_t$, eliminating the parameter from the learning. This simplifies the loss making constant the first term, leaving the second and third term.

For the second term, the KL divergence of two gaussian distributions can be computed in closed form as:

$$\mathcal{L}_2 = DL(q(x_{t-1}|x_t,x_0)||p_\theta(x_{t-1}|x_t) = \mathbb{E}_q\left[\frac{1}{2\sigma^2}||\tilde{\mu}_\theta(x_t,x_0) - \tilde{\mu}_t(x_t,x_0)||^2\right]$$

This reduces the computational cost of the estimation.

For the mean $\tilde{\mu}_t(x_t,x_0)$, if we consider that $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$ then, it can be rewritten as:

$$\tilde{\mu}_t(x_t,x_0) = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon\right)$$

This allows considering $\tilde{\mu}_\theta(x_t,x_0)$ as:

$$\tilde{\mu}_\theta(x_t,x_0) = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t,t)\right)$$

We can use a neural network for predicting the noise $\epsilon_\theta(x_t,t)$ of the process at step t given the sample x_t.

The original paper performs further simplifications to the loss observing that using strictly the loss obtained from the ELBO obtains samples of low quality. This simplification has the drawback of not allowing to compute exact likelihoods, since the ELBO is related to that quantity. The final loss drops the normalization constant and ends with the expression:

$$\mathcal{L}_2 = \mathbb{E}_{t,x_0,\epsilon}[||\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t},t)||^2]$$

We have still the last term of the ELBO loss. This term corresponds to a scaling of the data and a discretization decoder step from [-1,1] values to the range of the samples. For the case of images, the pixel values [0..255]. This is similar to what is done with VAEs or autoregressive models for images, the data is assumed gaussian and is then discretized to the pixels range to obtain a log likelihood for a discrete distribution. We can do the same in this case.

The algorithm for training is the following:

```
repeat
    Sample $x_0$ from $q(x_0)$
    Sample the time step $t$ from a uniform distribution $Uniform(\{1,\ldots,T\})$
    Sample the noise $\epsilon$ from $\mathcal{N}(0,1)$
    Apply gradient descent with
        $\nabla_\theta||\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon,t)||^2$
until Until convergence
```

When the noise network is trained we can sample with the algorithm:

```
Sample $x_T$ from $\mathcal{N}(0,1)$
for $t = 1 \ldots T$ do
    sample $z$ from $\mathcal{N}(0,1)$
    $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t,t)\right) + \sigma z$
end
```

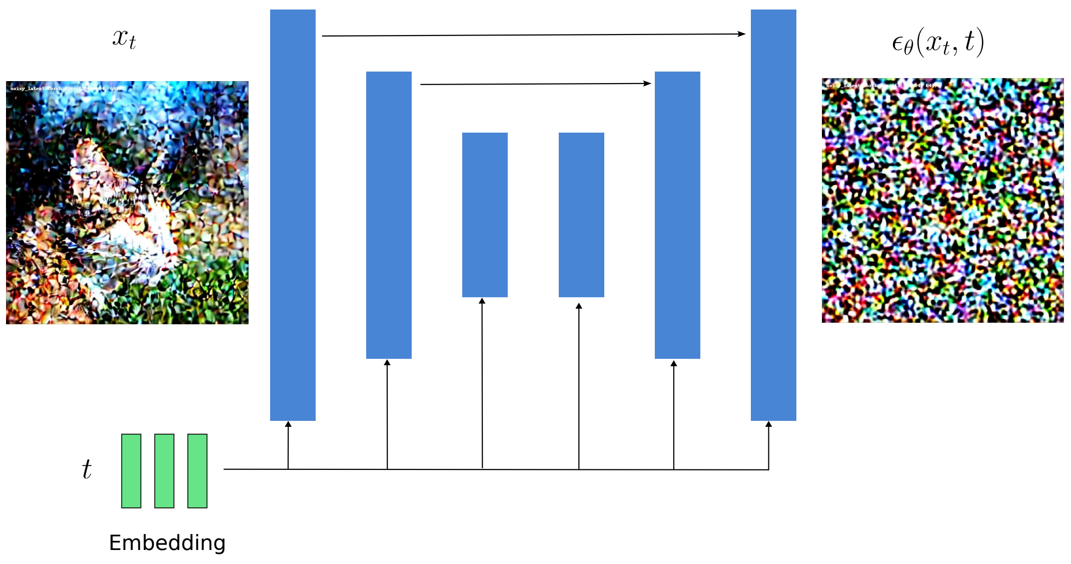


Figure 11.3: Schema of a U-Net architecture for noise prediction in DDPMs

Respect to the architecture that is used for predicting the noise the usual choice is a convolutional U-Net (see figure 11.3). This the same used for other tasks like image segmentation, and it is versatile. It has a structure similar to an autoencoder where each layer reduces the resolution by half until reaching a specific minimum resolution and then it is upscaled to the original dimensionality. The matching resolutions are connected by residual layers and the number of channels on each resolution is increased-decreased for each layer symmetrically. The residual connections allow for mixing at different scales and propagating gradients easily. A set of self attention layers is usually added to some resolutions.

The network has as inputs the noisy image on step t and the time step. This is introduced as conditioning for each layer using an embedding, usually a sinusoidal positional encoding, as in transformers. Only one network is trained for predicting the noise for all time steps, this reduces the number of parameters to be used and also helps to generalization.

The original DDPM made some simplifications and simple choices that worked in practice like using a noise schedule for training that followed a linear decay, also the noise during inference is fixed to the noise from training and the changes to the loss function do not allow computing exact log likelihood that is traded off for better sample quality. All these simplifications have been studied on successive methods like Improved DDPMs [100], Variational Diffusion Models [81] and Analytic DPM [11].

Some of these improvements include for instance different schedule functions for the noise that vary non-linearly like cosine decay or weightings for the loss terms based on the signal-to-noise ratio of the examples. Other improvements introduce the possibility of learning the adequate variances for the time steps during training using the ELBO or an analytic computation of mean and variance of noise computed after training to adequate it for inference. For obtaining better likelihood estimations also the loss has been modified to be able to combine the actual ELBO with the simplified version of DDPM with weights that depend on the time steps considering that early steps generate fine details (textures) and latter steps generate coarser details (image layout).

11.4 Noise Conditioned Score Networks (NCSNs)

We usually apply the Bayes formula to define the probability function we want to learn with a probabilistic model. For certain models, like the generative ones, the Bayes formula needs a normalizing constant for defining a proper probability distribution, this is also called the partition function. This normalization makes the function that represents the distribution to sum to one. This is usually intractable for many problems and makes difficult the optimization, since, for instance, the values of the function do not have a defined bound, or we can not compare distributions that need different normalizations.

In some cases we can work with unnormalized probability distribution functions, this is done using the **score function**. The score function is the gradient of the logarithm of the probability distribution function:

$$\nabla_x \log p(x)$$

The idea is that two distributions with the same score functions correspond to the same distribution (up to a normalization constant). If we can compute the score function for a dataset, then we can use it to match it with the distribution learned by a model.

We can use a neural network as our parameterized distribution s_θ, this is called a **score network**. The gradient of the network can be used to compare with the score function for the data. This can be trained using score matching techniques. The idea is that we can use the score function to sample from the probability distribution using a sampling technique that is called Langevin sampling. This method uses this function for generating samples. The sampling starts with a sample from a prior distribution and uses the recurrence:

$$x_t = x_{t-1} + \frac{\epsilon}{2}\nabla_x \log p(x_{t-1}) + \sqrt{\epsilon} z_t \;\; z_t \sim \mathcal{N}(0,1)$$

Noise Conditioned Score Networks [123, 124] propose to learn the score function using different levels of noise. The noise makes tractable the estimation of the score. The same network is used for all estimates as in DDPMs, the noise level is also an input used as conditioning, this has same role as the time step had in DDPMs. The loss that is used to train the network is:

$$\mathcal{L} = \frac{1}{2L}\sum_{i=1}^{L} \mathbb{E}_{p_{data}(x)} \mathbb{E}_{p_{\sigma_i}(\tilde{x}|x)} \left[\left|\left| \sigma_i s_\theta(\tilde{x}, \sigma_i) + \frac{\tilde{x} - x}{\sigma_i} \right|\right|_2^2 \right]$$

with $\tilde{x}$ noisy samples generated at different noise scales (increasing). There is a similarity between this loss and the one of DDPMs and the same training strategy can be used. We can obtain samples transformed with different levels of noise and train the network to predict the noise of consecutive steps.

The generating algorithm assumes that we have the set of increasing noise levels $\{\sigma_i\}_{i=1}^{T}$, a scaling factor ϵ for the sampling and a number of sampling steps T. The algorithm is as follows:

```
initialize $x_0$ as gaussian noise
for $i = 1 \ldots L$ do
    $\alpha_i = \epsilon \cdot \frac{\sigma_i}{\sigma_t}$
    for $t = 1 \ldots T$ do
        Sample $z$ from $\mathcal{N}(0,1)$
        $x_t = x_{t-1} + \frac{\alpha_i}{2} s_\theta(x_{t-1}, \sigma_i) + \sqrt{\alpha_i} z$
    end
    $x_0 = x_T$
end
return $x_T$
```

The algorithm starts with a gaussian sample and for the different levels of noise uses Langevine sampling for a number of steps guided by the score network following the gradient of the distribution a certain step size (α_i) at the current noise level. After passing through all noise levels the final sample should correspond to a sample from the data distribution.

11.5 Score-based generative modeling through SDEs

Score-based generative modeling through Stochastic Differential Equations [125] considers the forward process as continuous, so there is an infinite number of steps, and it is a process that depends on a continuous variable that is the time dimension.

In DDPM the process was defined as discrete markov chain with transitions that follow a gaussian distribution

$$q(x_t|x_{t-1} = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_t, \beta_t I))$$

then

$$x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\mathcal{N}(0, I)$$

defining β_t as a function of time with infinitesimal increments $\beta_t = \beta(t)\Delta t$ we can rewrite the process as the Stochastic Differential Equation (SDE)

$$x_t = \sqrt{1-\beta(t)\Delta t}x_{t-1} + \sqrt{\beta(t)\Delta t}\mathcal{N}(0, I)$$

if we apply Taylor series expansion to this equation we can approximate x_t by:

$$x_t \approx x_{t-1} + \frac{\beta(t)\Delta t}{2}x_{t-1} + \sqrt{\beta(t)\Delta t}\mathcal{N}(0, 1)$$

this defines the equation

$$dx_t = -\frac{1}{2}\beta(t)x_t dt + \sqrt{\beta(t)}d\omega_t$$

An Ordinary Differential Equation (ODE) is defined as:

$$dx = f(x, t)dt$$

This can be solved by integration from an initial condition $x(0)$ to a time step t

$$x(t) = x(0) + \int_0^t f(x, \tau)d\tau$$

We can apply a numerical ODE solver (like Runge-Kutta) to the equation

$$x(t + \Delta t) \approx x(t) + f(x(t), t)\Delta t$$

In this case this is a deterministic process and we will always obtain the same result from the same initial conditions.

Differently, a Stochastic Differential Equation (SDE) is defined as:

$$dx = f(x, t)dt + \sigma(x, t)d\omega_t$$

where $f(x, t)$ is called the drift coefficient, $\sigma(x, t)$ is called the diffusion coefficient and ω_t is Gaussian noise (a Wiener process). It can also be solved numerically with the equation

$$x(t + \Delta t) \approx x(t) + f(x(t), t)\Delta t + \sigma(x, t)\sqrt{\Delta t}\mathcal{N}(0, I)$$

This is a stochastic process that will obtain different results from the same initial conditions.

As you can see, the formulation of the process as an SDE is very similar to the two previous methods, but in this case the process is continuous, we can see the other methods as a particular case of this. The advantage of this formulation is that any SDE can be solved backwards in time, so we can obtain, starting from a sample in the final distribution, its initial conditions. That will correspond to a sample from the original data distribution. An additional advantage is that we can use any numerical solver for SDE for this task. We will have the forward diffusion SDE defined as:

$$dx_t = -\frac{1}{2}\beta(t)x_t dt + \sqrt{\beta(t)}d\omega_t$$

and the reverse diffusion SDE defined as:

$$dx_t = -\frac{1}{2}\beta(t)x_t dt - \beta(t)\nabla_{x_t}\log q_t(x_t) + \sqrt{\beta(t)}d\omega_t$$

where $\nabla_{x_t}\log q_t(x_t)$ is the score function.

As before, now the problem resides on computing a score function that allows applying the equation of the SDE. As before, we can use a neural network for this estimation. The probability density funcion $q_t(x_t)$ is not tractable, but we can condition to the data sample x_0 to obtain $q_t(x_t|x_0)$ that is tractable. In this case we can use the loss

$$\mathcal{L} = \mathbb{E}_t\left\{\lambda(t)\mathbb{E}_{x_0}\mathbb{E}_{x_t,x_0}\left[||s_\theta(x_t,t) - \nabla_{x_t}\log q_t(x_t|x_0)||_2^2\right]\right\}$$

where $\lambda(t)$ is a weighting function that depends on time and provides a different weight depending on the level of noise. This weight also appears on the other methods but is defined adhoc, in this case it appears directly on the formulation.

There is no need for changing the way of training of the score network, we can also sample the noise at different times for training with gradient descent. Once the network is trained, the reverse process uses its predictions for solving the SDE that we have defined. For this, the typical SDE solver algorithms can be used, for instance the Euler-Maruyama algorithm:

$$x_{t-1} = x_t + \frac{1}{2}\beta(t)x_t\Delta t + \beta(t)s_\theta(x_t,t)\Delta t + \sqrt{\beta(t)\Delta t}\mathcal{N}(0,I)$$

With this formulation, we are not limited to a specific noise schedule function $\beta(t)$, different choices will obtain a different diffusion processes. As we have pointed out before DDPM and NCSN correspond to special cases of this formulation where the noise function is discrete. It can be shown that DDPM is a process that maintains the variance of the process (variance preserving) and NCSN is a process that increases the variance (variance exploding). Also different choices for the weighting of the loss ($\lambda(t)$) allow different properties for the diffusion process, in particular it allows reproducing the simple loss from DDPMs or a loss that corresponds to the ELBO for better likelihood estimation.

These specific reverse time SDEs have an equivalent reverse ODEs, so we can perform deterministic mapping between samples and the noise distribution. The SDE and ODE estimate the same probability distribution when the ODE has the unit gaussian as target distribution. The corresponding ODE formulation is:

$$dx_t = -\frac{1}{2}\beta(t)x_t dt + \frac{1}{2}\beta(t)\nabla_{x_t}\log q_t(x_t)$$

This corresponds to a continuous normalizing flow, now solved using score matching, this connects both methods when the flow is defined as a continuous mapping.

The advantage of having a deterministic solution is to have options that the stochastic variant does not allow, for instance semantic interpolation of latent noises or sample editing by modifying the latent. Also, one of the problems of SDEs is that the score network has to be evaluated many times to obtain a sample, ODEs have more advanced numerical solvers that improve performance allowing to perform less evaluations of the score network. Moreover, being a normalizing flow (we start with gaussian noise) we can compute exact log likelihoods.

In summary, comparing both options, SDEs usually obtain samples of better quality, since the process can use corrections that improve the trajectory to the data distribution, but it is slower since often a fine-grained discretization is needed when solving the SDE, meaning more evaluations of the score network. ODEs samples have less quality, but are faster and allow operations on the latent space that are not possible with SDEs.

11.6 Accelerated sampling

Generative models have to solve what has been called the generative *trilema*. There are three qualities that we want to obtain:

1. To cover all the modes of the data distribution
2. To be able to generate samples fast
3. To generate samples of good quality

We can obtain 1 and 2 with flows and VAEs, we can obtain 2 and 3 with GANs, and we can obtain 1 and 3 with Diffusion Models. The question is if it is possible to accelerate Diffusion sampling, so we can have the perfect model.

The reverse step of diffusion uses noises of different levels for obtaining the samples, but not all the levels are equally important. It can be tested that the denoising focus on different structure of the image at different points. Skipping uniformly steps does not work since the noises do not have a uniform effect. One possibility is to learn a noise schedule that only uses the noises that are important for good quality samples.

11.6.1 Accelerating the forward process

Denoising Diffusion Implicit Models [122] breaks with the assumption that the forward and reverse processes are markovian. The inference distribution used by this model adds a conditioning on x_0 for obtaining a tractable distribution. The distribution is:

$$q(x_{1:T}|x_0) = q(x_T|x_0)\prod_{t=2}^{T} q(x_{t-1}|x_t, x_0))$$

The mean of the distribution $q(x_{t-1}|x_t, x_0))$ is chosen, so it matches the one in DDPMs on the steps of the forward process:

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I)$$

This formulation allows defining distributions for intermediate steps in the forward process, so not all the steps are necessary. This means that we can also jump time steps computing the reverse process. The distributions between jumps are chosen so in intermediate states, the marginal distribution conditioned to x_0 matches the distributions defined by DDPM. The training loss is the same as in DDPMs, so a pretrained DDPM model can be used for inference using the alternative distribution for obtaining the noise at specific steps of the process. From the experiments, with a number of steps in the range of 1000 in the forward process, good results can be obtained performing one of every 5 or 10 steps instead of all the steps.

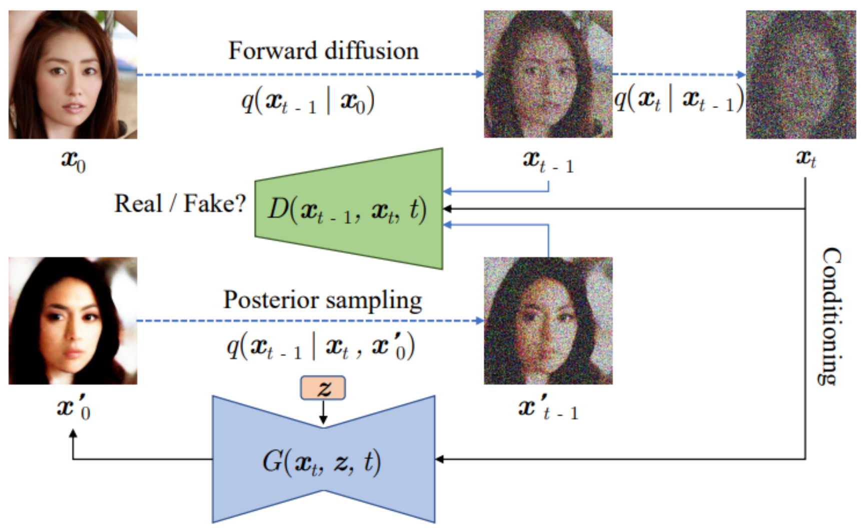

Forward diffusion
$q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_0)$
$q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1})$
$\boldsymbol{x}_0$
$\boldsymbol{x}_{t-1}$
$\boldsymbol{x}_t$
Real / Fake?
$D(\boldsymbol{x}_{t-1}, \boldsymbol{x}_t, t)$
Conditioning
Posterior sampling
$q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t, \boldsymbol{x}'_0)$
$\boldsymbol{z}$
$\boldsymbol{x}'_0$
$\boldsymbol{x}'_{t-1}$
$G(\boldsymbol{x}_t, \boldsymbol{z}, t)$

Figure 11.4: Denoising Diffusion GAN training schema

11.6.2 Accelerating the reverse process

The reverse process depends on simple unimodal distributions for each transition (usually gaussian). To skip steps in the reverse process we need more complex distributions so they can model complex changes since the changes do not correspond to unimodal distributions any more. We also will use the same model for all the noise predictions, but the predictions have to be more simple or complex depending on the point in the process, since the behavior is different. Diffusion processes also depend on the dimensionality of the data, we could embed the data to a smaller space and apply the diffusion on that space so we can model better the changes.

One proposal for accelerating the reverse process by defining more complex distributions is Denoising Diffusion GANs [138]. The idea is to use a conditioned GAN combined with the diffusion model. A GAN is able to capture multimodal distributions that is what is needed for performing larger steps on the reverse process. The GAN uses the U-Net as generator adding a latent introduced after the encoder of the network. The GAN applied is the non saturating GAN, but using the softened reverse KL. The discriminator decides if one image is the next plausible diffusion step of other image, contrasting samples generated by the diffusion model and real pairs of noisy images. Figure 11.4 shows the training schema of this model. The forward diffusion process is used for obtaining samples noised at different steps, the U-Net acts as a GAN generator that obtains the feedback of a discriminator that determines if two images correspond to real or generated denoising.

An alternative is to learn a network that compresses the steps of the reverse process. This is the approach proposed by [117]. The idea is to transfer the denoising network to another network using distillation. Distillation is a training technique that involves a teacher and a student network. The student network learns to reproduce the output of the teacher.

In this case the architecture of the teacher and the student are the same. The student starts learning to reproduce the output of applying two consecutive denoising steps computed by the teacher. Once the student has converged the student becomes the teacher and a new student network is trained using the same method. This is repeated until reaching a specific reduction in the number of steps.

One problem with this distillation process is that the same loss used for training can not be used for the distillation. The main problem is that as the number of steps predicted is increased

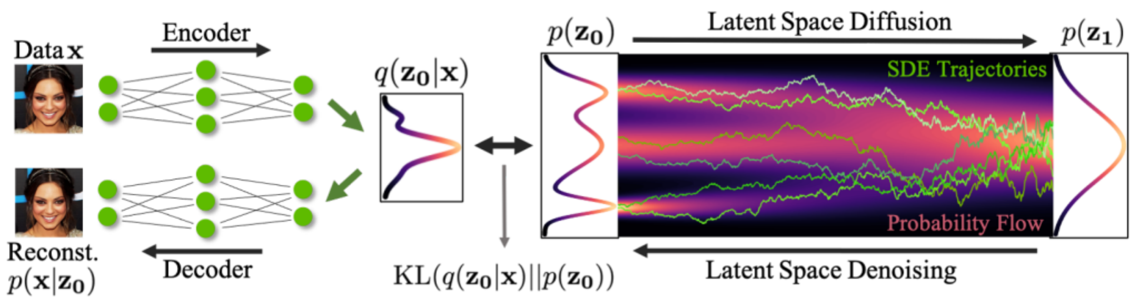


Figure 11.5: Process of denoising using a VAE as latent representation

there is a mismatch between the images from two consecutive steps, for instance, in the extreme we go from pure noise to the real image. Basically, the noise to signal ratio between the images changes with the steps.

The paper proposes different losses that can be used for distillation that will be stable when the SNR changes, like predicting also the original image, predicting the image and the noise using different channels of the network and then combining them or predicting an affine transformation with the noise levels of consecutive steps and the original image.

Another dimension that can be used to scale the reverse process is to perform it in a lower dimensional latent space. This will reduce the computational cost of the process maintaining all the steps of the denoising and also will allow generating samples at larger resolutions, since the diffusion network will depend on the dimensions of the latent space. This is the proposal in Latent Diffusion [129] that uses a VAE to transform the original samples to a latent space that follows a gaussian distribution. With the transformation we also address the problem of having samples that do not follow a gaussian distribution. The diffusion process makes this assumption (gausianity of the data), and now it holds on the latent space since we are forcing to be gaussian using the VAE.

Now the denoising diffusion model will be the prior that we can use with the decoder of the VAE. We can transform gaussian noise in the latent space to samples that are in the VAE learned latent space and then transform them to data samples using the decoder.

In this model we are jointly training two models, the VAE and the Denoising Diffusion model. The VAE encoder transforms examples to the latent space, the latent values are used for the diffusion model, that has to match their distribution to the one generated with the reverse process and the latent samples are transformed by the decoder to the original sample. The loss correspond to the ELBO combining the distributions of the model. Figure 11.5 represents this process.

11.7 Conditioning

As in previous models, we could be interested on using additional information for the sample generation for:

- Generating class specific examples (when classes are known)
- Performing reconstruction from partial information (e.g. inpainting/outpainting)
- Manipulating samples (e.g. image editing, colorization)
- Performing multimodal translation like image to image translation, text to image generation, speech to text...

Differently from other models like GANs or Flows, we can not manipulate the input noise (the latent) for conditioning, since changing the noise will return a different sample, not a variation of the sample that corresponds to that latent.

In these models we have to change the distribution of the reverse process, so the path that is followed by the reverse process is steered by the additional information of the conditioning. The denoising step will follow a distribution defined as $p(x_{t-1}|x_t,c)$ where the side information c is part of the conditional distribution. The distribution of the model will be

$$p_\theta(x_{0:T}|c) = p(x_T)\prod_{t=1}^{T} p_\theta(x_{t-1}|x_t,c)$$

where $p_\theta(x_{t-1}|x_t,c) = \mathcal{N}(x_{t-1};\mu_\theta(x_t,c),\Sigma_\theta(x_t,c))$. This only changes the input that the denoising network will receive including the conditional information. Basically the U-Net architecture will be adapted depending on the nature of the conditioning:

- If it is a scalar, for instance, a class, it can be incorporated as an additional embedding at the input or by applying layer normalization with the embedding for each layer
- If it is an image, for instance for inpainting or segmentation mask, this can be incorporated as a channel-wise concatenation with the data or by using cross attention with some layers using an embedding of the conditioning (to reduce size)
- If it is text, for instance in text to image models, this can be incorporated as single vector embedding with scalar addition to the time embedding, or with a sequential embedding applying cross attention with different layers

11.7.1 Guidance

An alternative for class conditioning is classifier guidance, presented in [30]. The idea is to use an auxiliary classifier able to determine the class of noisy images. For each denoising step, the input image is passed through the classifier and the gradient that is obtained is used to steer the sample towards the class. The idea is to change the direction of the reverse process, so it is directed to the part of the data space where the samples that are from the class reside. The obvious drawback is to have to train an additional classifier that is able to determine the class of a sample even when it is noisy.

To avoid this problem in [67] a classifier free alternative is defined. The advantage of this is that the conditioning is not limited to a class, it could be any other information and this is the method that is used with most of the text-to-image diffusion models. The idea is to compare the noise that is predicted by the score network when there is and there is not conditioning. With this we can obtain an implicit classifier considering that

$$p(c|x_t) \propto p(x_t|c)/p(x_t)$$

For obtaining the score network only one model is trained. During training the network has as input the actual conditioning and, randomly, the conditioning that is passed is a vector of zeros, considering this vector the empty conditioning. In the end it is like to have a special conditioning for all the samples.

For denoising an image on a step, a combination of the noise with and without conditioning is used, where the parameter of the combination is a hyper-parameter that we can change to obtain more or less strength in the conditioning. Changing the value is a trade-off between diversity and quality, the larger the weight the higher the quality (closer to the class or more faithful to the conditioning information). The noise is defined as:

$$\tilde{\epsilon}_\theta(x_t,c) = (1+w)\epsilon_\theta(x_t,c) - w\epsilon_\theta(x_t)$$

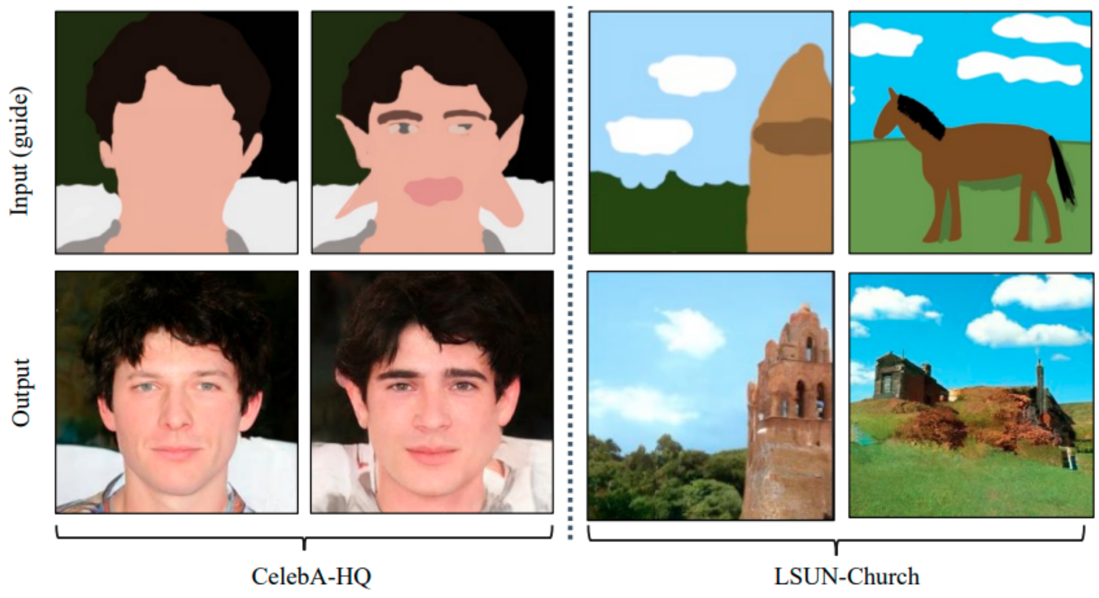


Figure 11.6: Transforming and editing brush stroke images with the SEdit model (source [92])

11.7.2 Complex conditioning

Conditioning is the basis of many models that perform different kind of image to image translation. For instance, the model presented in [114] uses an improved version of DDPM to perform image super resolution using a cascade of diffusion models. Each one doubles the resolution of the previous one. In this case we have to train score networks for each resolution. The model is trained using as conditioning images at the specific resolution. On inference time the up-scaled sample generated by the previous model is used for the conditioning of the next stage. We can start with a gaussian noise in the first resolution, the next one uses up-scaled gaussian noise and the up-scaled generated image of the first stage, and we can iterate this until reaching our target resolution (the one of the last trained model). The weight of the conditioning has to be enough for obtaining the same image (with finer detail) on each stage.

The model Palette [115] describes a diffusion model for different tasks including colorization, conditioning to black and white images, inpainting, conditioning to a mask, uncropping/outpainting, conditioning to incomplete images and restoration, conditioning to damaged images.

The model SEdit [92] uses denoising diffusion for image to image translation from brush stroke paintings (similar to what is possible with pix2pix). Brush stroke paintings define a rough sketch of the content of the image. There are different brushes for different content and it could be considered a segmentation mask. That image can be perturbed using the forward process until reaching a specific time step to get it closer to the distribution of the real data. Then, the reverse process can be applied to obtain a real image. This can be used for image editing modifying the initial brush stoke painting and applying the forward and reverse process with the modified image. There are also models based on GANs that also allow image editing, but they involve the inversion of the image to obtain the corresponding latent for performing modifications. That needs for an optimization step that is expensive, so this model is computationally cheaper. Figure 11.6 shows examples of transformations of brush stokes paintings transformed to real images with models trained with the CelebA and LSUN-Church datasets. On the left, the initial image is edited obtaining a similar final image, on the right, even when the initial image makes no sense for the dataset, the model is able to generate a consistent image.

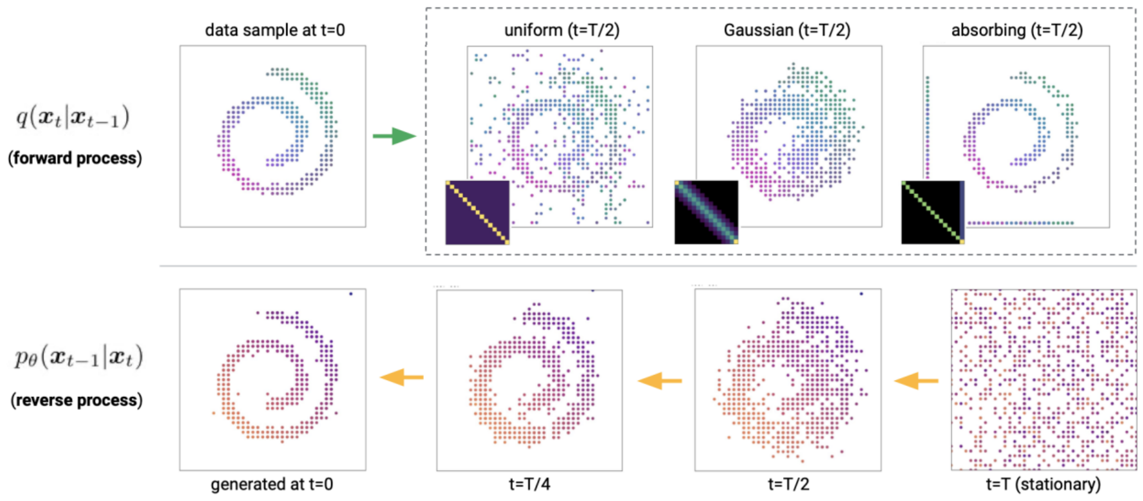


Figure 11.7: Discrete diffusion forward processes with uniform, discrete gaussiand and absorbing state transition matrices (source [8])

11.8 Denoising Diffusion for discrete data

All the previous diffusion models assume that the data is continuous and that the diffusion process is defined using a continuous distribution (gaussian). There are many domains where the data is discrete, for instance tabular data or natural language, where words are represented as tokens. We have also seen that there are models that are able to quantize the data to a discrete representation using Vector Quantization, with the advantage of reducing the dimensionality of the data and improving scalability.

In order to work with these data we have to define the diffusion process for discrete distributions. One way for working with categorical distributions is to use a one hot encoded representation. With this we can define a transition matrix between values that will correspond to a markov process. Now the diffusion process will be discrete, and the transition matrix will define the probability of the transitions.

The transition matrix can describe different forward processes. As you recall, the purpose of the forward process is to progressively destroy the data, so the transitions will change the original values of the data to other random values until the sample is basically noise. We can for instance define transitions defined by a uniform categorical distribution, so all categories have the same probability and any initial value has the same probability of ending on any other value. Alternatively, we can use a continuous distribution as transition probabilities but discretized, so different categories will have different probabilities. Another possibility is to have transitions that have a probability for ending in an absorbing state, a new value that corresponds to a special state that can not change. This final process corresponds to a progressive masking of the data, from the initial sample to a sample where all is masked. Figure 11.7 shows at the top a sample in the mid of the forward process defined by these three alternatives, a uniform distributed, a discretized gaussian and an absorbing state transition matrix, The reverse process will start with a noisy sample and the score matching network will define the backward transitions. Figure 11.7 shows at the bottom a sample that starts as uniform noise and is transformed to the data distribution.

The possibility of using an absorbing state transition matrix that behaves like masking samples links this models to language models. Language models, like for instance BERT, are trained for predicting masked samples. A sequence of tokens that correspond to a sentence has one or several tokens substituted by a special token (mask) and the model has to predict

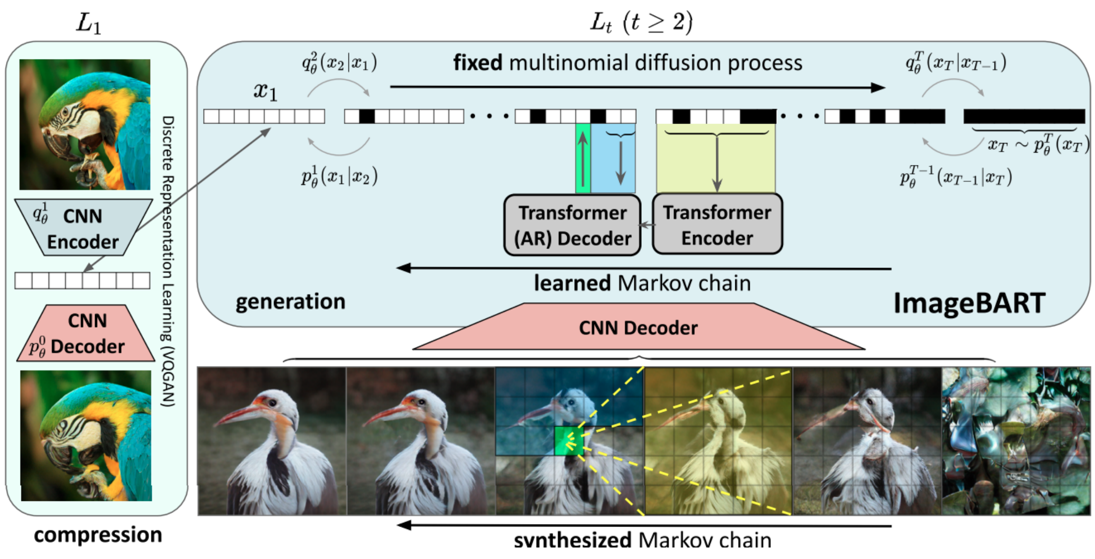


Figure 11.8: Training process and architecture of ImageBART (source [41])

the values that have been masked. This corresponds to a one-step discrete diffusion model. In general, the most known language models can be interpreted as discrete diffusion models that use masking. This opens the possibility of using transformers as the score matching network for discrete diffusion models not only for discrete data, but also for images.

ImageBART [41] defines a model that uses Vector Quantization as in other previous models for obtaining a representation of images with a discrete vocabulary. The quantization is obtained with an auto encoder that is based on VQ-GAN. This is a vector quantizer that works at patch level using a loss that combines a perceptual loss (LPIPS) that makes sure that the features learned correspond to realistic images by comparing the features learned to the features of a network pretrained on Imagenet, and an adversarial loss that makes sure that patches correspond to the data.

After the quantization process, any image can be transformed to a sequence of tokens from the vocabulary followig the image raster order. The forward diffusion process uses a discrete transition distribution with an absorbing state that masks progressively elements of the sequence until all is masked. The reverse process is learned by a transformer network that fits an autoregressive distribution that unmasks the sample. The autoregressive distribution unmasks one by one the elements of the sequence using at each step the previous sequence. This makes the number of steps to be performed for unmasking an image equal as the number of patches defined by the image resolution and the quantization level. From the denoised samples the decoder can be applied to obtain the corresponding image. Figure 11.8 represents the architecture and the training process of the model. The Encoder-Decoder model transforms into tokens an image, the forward process obtains the masked samples that defines the autoregressive distribution that is learned by the transformer model. The Encoder-Decoder and the diffusion model can be learned separately. This model can also be conditioned adding more tokens to the representation that correspond to the conditioning information, just like it is done with other transformers models.

Maskgit [20] goes a step further. Autoregressive distributions are not really adequate when there are complex dependencies among the variables of the data. On image generation, pixels/patches depend on information of the local context. In this model, the same autoencoder architecture is used for quantizing the images and learning a dictionary of tokens. The main difference is that the prediction of the tokens is done for set of T steps, where at each step, all tokens are predicted by the transformer from a masked image, and only the most confident ones are fixed. Then the other tokens are masked, and the process is repeated for T steps until all tokens are predicted. The number of tokens to fix at each step are a function of the number of steps and the total number of masked tokens in the image. Only the tokens with less confidence may be masked.

The model for unmasking is a bidirectional transformer. Since the unmasking is not autoregressive this model is faster, all tokens are predicted in parallel, and the computational cost depends on the number of steps that is used, that is a hyperparameter.

11.9 Notebook: Denoising Diffusion Models

In this notebook we will work with diffusion models. As you will remember, these models are based on reversing the process of progressively adding noise to an example, considering that the number of steps used is long enough so that it can be modeled as a Markovian process.

The models are trained so that a network is able to predict, given the point in the process, what noise was added to an example, so that it can be eliminated. The usual model is a U-Net network that is formed by an encoder plus a decoder with connections between them at matching resolutions.

We will use the Hugginface `diffusers` library and some auxiliary libraries

11.9.1 Diffused numbers

We will again use the MNIST dataset using the first 10000 examples. We will also use the same dataloader.

[5]:
```
dsize = 10000
mnist_trainset = datasets.MNIST(root='../../data', train=True,␣
 ↪download=True, transform=None)
train_dataset = (mnist_trainset.data[:dsize].reshape(-1, 1, 28, 28) / 127.)␣
 ↪-1.
train_labels =  mnist_trainset.targets[:dsize]

class MNIST(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.label = labels

    def __getitem__(self, index):
        x = self.data[index]

        return self.data[index], self.label[index]

    def __len__(self):
        return len(self.data)
dataset =  MNIST(train_dataset, train_labels)

mnist_dl = torch.utils.data.DataLoader(dataset, batch_size=256)
```

To train the diffusion model we will need different elements. The first is the network that learns to predict the noise that a sample has at a given moment in the diffusion process.

The diffusers library has a series of blocks that we can combine to build the network. We will generally assume that the network is a convolution-based U-Net. The basic blocks of the network correspond to residual convolutional networks (ResNet).

For the U-Net architecture we have a part of the network that encodes by halving the size of the input to each layer and a part that decodes by doubling the size of the input to each layer. These groups of blocks are connected by *skip connections* at the matching resolutions and the central part of the network corresponding to the minimum resolution usually has twice as many channels as the previous layer. The Cross Attention layer is connected to this layer if the model is conditioned to something more complex than the class.

In this case we will define a network that will have 28*28 images with a single channel as input, and at the output we will have the same size and number of channels. We will have two ResNet blocks for the encoder and two for the decoder (always the same number on both sides). We define how many channels the output of each block will have and if we want more than one layer for each block. The parameters have been adjuster *manualy*.

[6]:
```
model_params = {"sample_size": 28,
            "in_channels": 1,
            "out_channels": 1,
            "down_block_types": [
                "DownBlock2D",
                "DownBlock2D",
            ],
            "up_block_types": [
                "UpBlock2D",
                "UpBlock2D"
            ],
            "block_out_channels": [
                32, 64
            ],
            "layers_per_block": 1
}
```

We define a U-Net with these parameters

[7]:
```
model = UNet2DModel(**model_params)
```

As you will remember, the network must estimate the noise for a process that must be long enough to correspond to a Markov chain. This is defined in the variable `training_steps` which we will set to 1000. This means that to transform Gaussian noise to an image we will do 1000 inference steps with the network until we reach an image.

We also define the number of training epochs (you can imagine that training will take a long time if the number is high)

[8]:
```
epochs = 10
training_steps = 1000
```

The second element we need is a function that determines the evolution of the noise (scheduler), basically what decides how large is the variance of the noise that is added to the samples. You need to know how long the process is, what prediction the network is going to make and how the noise variance is going to decay.

The function has a default initial and final variance that we will leave as is. The noise reduction will be linear with the steps of the process and we will define a network that will predict the noise in a step of the process (you can look in the documentation to see what other possibilities and parameters there are)

[9]:
```
noise_scheduler = DDPMScheduler(num_train_timesteps=training_steps,
                                beta_schedule="linear",
                                prediction_type= "epsilon"
                                )
```

We will obviously need an optimization algorithm (its parameters would have to be adjusted) and we define a scheduler for the learning rate

[10]:
```
optimizer = optim.Adam(model.parameters(), lr=1e-3)

lr_scheduler = get_scheduler(
        "cosine",
        optimizer=optimizer,
        num_warmup_steps=100,
        num_training_steps=(len(mnist_dl) * training_steps),
    )
```

This is the training loop. We do it for a number of epochs and in each epoch we go through the entire training set doing:

- We obtain a batch of images
- We obtain a batch of Gaussian noise
- We obtain a batch of time steps of the diffusion process uniformly in the range of steps
- We generate the noisy images corresponding to the time steps according to the noise scheduler
- We predict noise with the U-Net
- We calculate the MSE between the prediction and the noise
- We reset the network

We will use the `accelerate` framework to manage the training of the network, basically it is responsible for simplifying the code that must be written in the training loop by managing the movement of data and model to the GPU, the backward step and other more complex things, like multi GPU training (which we won't use)

[11]:
```
def train_loop(model, noise_scheduler, training_steps, dataloader,␣
 ↪optimizer, lr_scheduler, epochs, cond=False):
    # Accelerate manages the training elements
    acc = Accelerator()
    network, opt, train_data, lr_sched = acc.prepare(model, optimizer,␣
 ↪dataloader, lr_scheduler)

    for epoch in tnrange(epochs):
        for  batch in train_data:
            # 1- Images Batch
            clean_images = batch[0]

            # 2- Gausian noise Batch
            noise = torch.randn(clean_images.shape).to(clean_images.device)

            # 3- Time steps batch for images generated uniformly
            bsz = clean_images.shape[0]
            timesteps = torch.randint(0, training_steps, (bsz,),␣
 ↪device=clean_images.device).long()
```

```
            # 4- We apply the forward diffusion step to each image until the
            #    corresponding timestep
            noisy_images = noise_scheduler.add_noise(clean_images, noise,␣
↪timesteps)

            # 5- We obtain the noise prediction from the network
            if cond:
                model_output = network(noisy_images, timesteps,␣
↪class_labels=batch[1]).sample
            else:
                model_output = network(noisy_images, timesteps).sample

            # 6- Compute the MSE between the predicted noise and the real␣
↪noise
            loss = F.mse_loss(model_output, noise)

            # 7- Backpropagate the loss
            acc.backward(loss)
            acc.clip_grad_norm_(network.parameters(), 1.0) # Gradient is␣
↪clipped for more stable training
            opt.step()
            lr_sched.step()
            opt.zero_grad()
    acc.free_memory()
    return acc.unwrap_model(network)
```

Now we train for a while (if we are not going to do enough steps we can leave save_mode to False and let the pre-trained model load)

[12]:
```
save_model = False
```

[13]:
```
unet = train_loop(model,
                  noise_scheduler,
                  training_steps,
                  mnist_dl,
                  optimizer,
                  lr_scheduler, epochs)
```

[14]:
```
if save_model:
    torch.save(unet.state_dict(), '/content/Models/MNIST-DDPM.pt')
else:
   unet = UNet2DModel(**model_params)
   unet.load_state_dict(torch.load('/content/Models/MNIST-DDPM.pt'))
   unet.eval()
   unet.to(device);
```

In order to sample the model, the `diffusers` library has a series of functions that facilitate the entire process, they only need the network and a scheduler for the noise (it does not have to be the same one with which the model was trained)

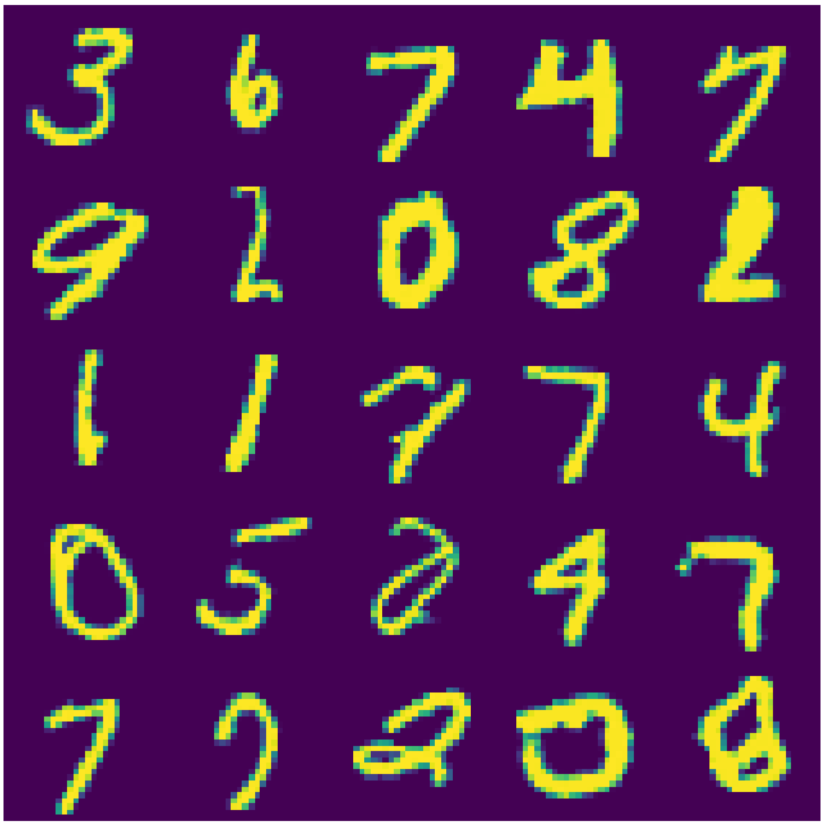
[15]:
```
pipeline = DDPMPipeline(
    unet=unet,
    scheduler=noise_scheduler,
)
```

Now we can generate images with the `pipeline` that the scheduler and the network will use to obtain images from Gaussian noise. The variable `sampling_steps` determines the number of steps in the process, which does not have to coincide with the one used in training. Depending on the scheduler, we will be able to obtain good images even though we do not go through the entire process, but a minimum of steps will be required.

[16]:
```
sampling_steps = 1000
generator = torch.Generator(device=pipeline.device)
nim = 5
samples = pipeline(
    generator=generator,
    batch_size=nim*nim,
    num_inference_steps=sampling_steps,
    output_type="numpy"
).images
```

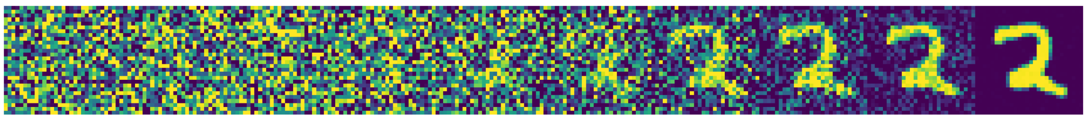
Depending on the number of steps in the denoising process (and how much we have trained the model we will be able to see something that makes sense)

[17]:
```
fig = plt.figure(figsize=(5,5))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(samples[i])
    plt.axis('off')
plt.tight_layout(pad=0.)
```



We can observe how the process is done from the noise to the image by reproducing the algorithm that does the sampling following the diffusion process. We simply have to start with random noise, and iterate by predicting the noise with the network and letting the scheduler eliminate the noise according to the moment of the process we are in. We can see how the image appears little by little.

[18]:
```
@torch.no_grad()
def denoise(unet, dims, factor):
    image = randn_tensor(dims, generator=generator, device=device)
    proc = []
    for t in reversed(range(training_steps)):
        if (t%factor)==0:
            proc.append(np.clip((image.detach().cpu().permute(0,2,3,1).
 ↪numpy() / 2 + 0.5),0,1)[0])
        model_output = unet(image, t).sample
        image = noise_scheduler.step(model_output, t, image,␣
 ↪generator=generator).prev_sample
    return proc

factor = 100
dims = (1,1, 28,28)
proc = denoise(unet, dims, factor)

fig = plt.figure(figsize=(20,5))
for i, im in enumerate(proc):
    plt.subplot(1, 10, i+1)
    plt.imshow(im)
    plt.axis('off')
plt.tight_layout(pad=0.)
torch.cuda.empty_cache()
```



11.9.2 Generation quality

We can also calculate the quality of the data we generate. As we have seen in class on the topic of GANs, there are some quality measures that can be applied that obtain features from a set of images using a pre-trained network. The Frechet Inception Distance measure uses an InceptionV3 network, calculates features for a real data set and generator, and makes a calculation with the distribution of these features.

Let's calculate this measure for the data we have generated. Obviously this measurement can only be used to compare different generators. What we are going to do is compare how a sample of the generator that we have trained looks like with some of the data that we have not used for training (let the more real data win!).

We have an implementation of this measurement in the `torchmetrics` library.

[19]:
```
from torchmetrics.image.fid import FrechetInceptionDistance
```

Now we generate 1000 samples with our generator

[20]:
```
sampling_steps = 1000
generator = torch.Generator(device=pipeline.device)
images = pipeline(
    generator=generator,
    batch_size=1000,
    num_inference_steps=sampling_steps,
    output_type="numpy"
).images
```

We make three dataloaders, one for the real data with which we have trained, others for unused real data and others for the generated data. Since the data only has one channel and the Inception network expects three, we simply triple it.

[21]:
```
fake_dl = torch.utils.data.DataLoader(torch.tensor(images*255, dtype=torch.
 ↪uint8).permute(0,3,1,2).expand(-1,3,-1,-1), batch_size=64)
sub_dataset = torch.tensor(train_dataset[:1000].reshape(-1, 1, 28, 28).
 ↪numpy()*255, dtype=torch.uint8)
sub_dataset2 = mnist_trainset.data[dsize:dsize+1000].reshape(-1, 1, 28, 28)


real_dl = torch.utils.data.DataLoader(sub_dataset.expand(-1,3,-1,-1),␣
 ↪batch_size=64)
real_dl2 = torch.utils.data.DataLoader(sub_dataset2.expand(-1,3,-1,-1),␣
 ↪batch_size=64)
```

We calculate it by comparing real with generated (`feature` is the size that corresponds to the Inception layer that will be used to calculate the features)

[22]:
```
fid = FrechetInceptionDistance(feature=192).to('cuda')

from tqdm import tqdm
for batch in tqdm(real_dl):
    fid.update(batch.to(device), real=True)

for batch in tqdm(fake_dl):
    fid.update(batch.to(device), real=False)

print('FID=',fid.compute() )
```

```
FID= tensor(0.5635, device='cuda:0')
```

Now real against real

[23]:
```
fid = FrechetInceptionDistance(feature=192).to('cuda')

from tqdm import tqdm
for batch in tqdm(real_dl):
    fid.update(batch.to(device), real=True)

for batch in tqdm(real_dl2):
    fid.update(batch.to(device), real=False)

print('FID=',fid.compute() )
```

```
FID= tensor(0.5703, device='cuda:0')
```

If the difference is not very large, we can consider that the generated data has a *reasonable* resemblance to the real ones.

[24]:
```
torch.cuda.empty_cache()
```

Another possible way to see the diversity of the data is to use a near neighbor algorithm to see how similar the generated samples are to the training data (obviously we can only do that for a few). To do this we will index the training data and search for some of the generated samples.

[25]:
```
from sklearn.neighbors import KDTree
```

We index the training data into a KDTree

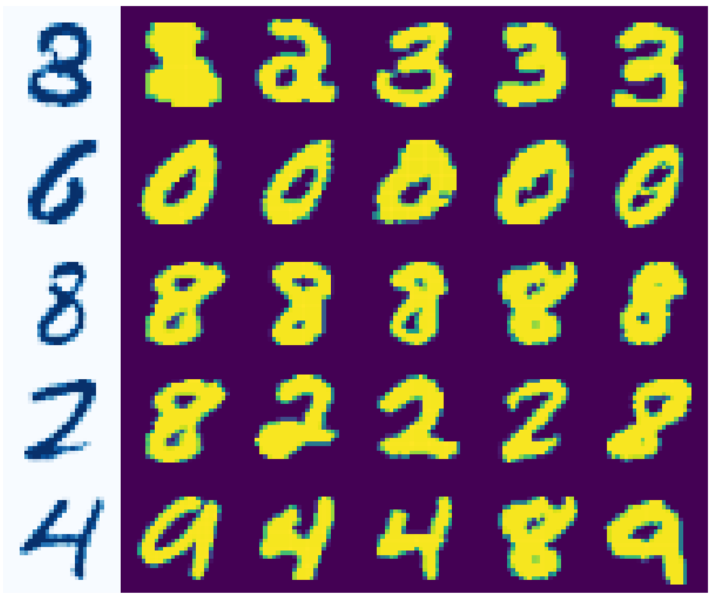
[26]:
```
real_data = (train_dataset.numpy().reshape(-1, 28*28)*2)+1
kdtree = KDTree(real_data)
```

And now we look at the neighbors who are closest and their distance

[27]:
```
fig = plt.figure(figsize=(6,5))
nimg = 5
nknn = 5
p = 1
for i in range(nimg):
    neigh = kdtree.query((images[i].reshape(-1, 28*28)*2)+1, k=nknn)
    plt.subplot(nimg, nknn+1, p)
```

```
        plt.imshow(images[i].reshape(28,28,1),cmap='Blues')
        plt.axis('off')
        for j in range(nknn):
            plt.subplot(nimg, nknn+1, p+j+1)
            plt.imshow(real_data[neigh[1][0][j]].reshape(28,28,1))
            plt.axis('off')
        p+=(nknn+1)
        print(f"Distancia media {i} = {neigh[0][0][i].mean()}")
plt.tight_layout(pad=0.)
```

```
Mean Distance 0 = 50.20277370201591
Mean Distance 1 = 50.151922255649154
Mean Distance 2 = 50.8539185264915
Mean Distance 3 = 51.63356161683394
Mean Distance 4 = 52.42641636848645
```



11.9.3 Conditioning

With this model we cannot know what it will generate, we can generate a model conditioned on the classes in a similar way to how we did it in GANs. For the simple case of conditioning on a class we have different options, the one we are going to use directly introduces an embedding of the class identifier and adds it to the positional embedding that is used to represent the temporal instant of the diffusion process. The network will be the same as before, simply adding a linear layer that will learn the embedding of the classes.

[28]:
```
model_params_c = {"sample_size": 28,
            "in_channels": 1,
            "out_channels": 1,
            "down_block_types": [
                "DownBlock2D",
                "DownBlock2D",
            ],
            "up_block_types": [
                "UpBlock2D",
                "UpBlock2D"
            ],
            "block_out_channels": [
                32, 64
            ],
            "layers_per_block": 1,
            "class_embed_type": "timestep",
            "num_class_embeds": 10
}
```

We define the network

[29]:
```
model_c = UNet2DModel(**model_params_c)
```

We use the same optimizer

[30]:
```
optimizer = optim.Adam(model_c.parameters(), lr=1e-3)
lr_scheduler = get_scheduler(
        "cosine",
        optimizer=optimizer,
        num_warmup_steps=100,
        num_training_steps=(len(mnist_dl) * training_steps),
    )
```

The training loop that we are using is already prepare for conditional training

[31]:
```
unet_c = train_loop(model_c,
                  noise_scheduler,
                  training_steps,
                  mnist_dl,
                  optimizer,
                  lr_scheduler, epochs, cond=True)
```

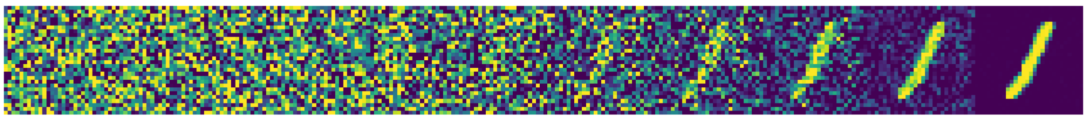
[32]:
```
save_model = False
if save_model:
    torch.save(unet_c.state_dict(), '/content/Models/MNIST-DDPM-c.pt')
else:
    unet_c = UNet2DModel(**model_params_c)
    unet_c.load_state_dict(torch.load('/content/Models/MNIST-DDPM-c.pt'))
    unet_c.eval()
    unet_c.to(device);
```

Now we can generate the samples given a class. We define a function identical to the previous one, but now it supports the class

[33]:
```
@torch.no_grad()
def denoise_cond(unet, dims, factor, label):
    generator = torch.Generator(device=device)
    image = randn_tensor(dims, generator=generator, device=device)
    proc = []
    for t in reversed(range(training_steps)):
        if (t%factor)==0:
            proc.append(np.clip((image.detach().cpu().permute(0,2,3,1).
↪numpy() / 2 + 0.5),0,1)[0])
        model_output = unet(image, t, torch.tensor([label]).to(device)).
↪sample
        image = noise_scheduler.step(model_output, t, image,␣
↪generator=generator).prev_sample
    return proc
```

and we see how the image is generated, we just have to change the parameter that corresponds to the class

[34]:
```
factor = 100
dims = (1,1, 28,28)
proc = denoise_cond(unet_c, dims, factor, 1)
fig = plt.figure(figsize=(20,5))
for i, im in enumerate(proc):
    plt.subplot(1, 10, i+1)
    plt.imshow(im)
    plt.axis('off')
plt.tight_layout(pad=0.)
torch.cuda.empty_cache()
```



11.9.4 Fast generation

As we saw in class, the DDPM model defines the generation process in such a way that there must be many denoising steps when doing inference to obtain quality samples. We can see the difference with other models using a pre-trained model on the CIFAR10 dataset (32x32 images of 10 classes)

We are going to do 50 inference steps only (of the 1000 used for training) to generate samples using:

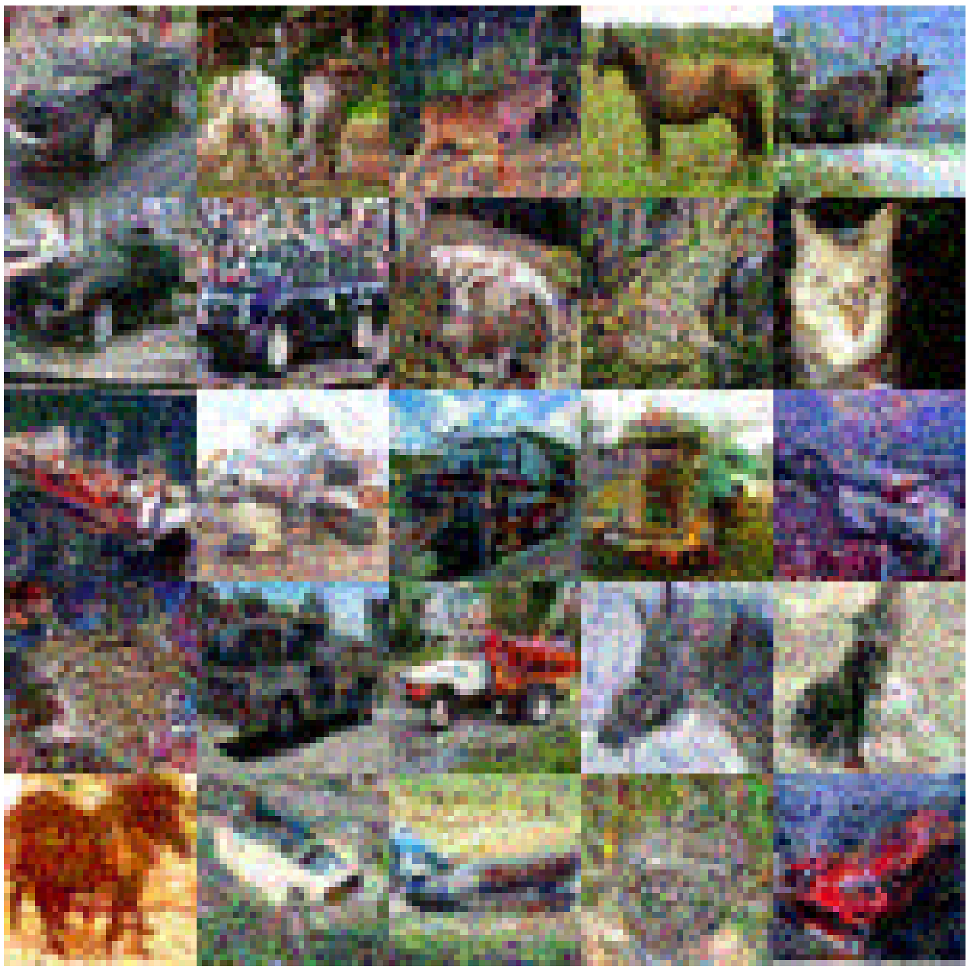
- DDPM (which assumes that we follow the markov process step by step)
- DDIM (based on defining a deterministic process that is equivalent to DDPM in result and allows jumping a certain number of steps in each iteration, we make jumps of equal length)
- PNDM (based on solving the problem as differential equations so that we can estimate how much we can jump in the continuous process to the next step)

Obviously the closer we get to the DDPM training steps it will generate higher quality images.

[35]:
```
inference_steps = 50
```

We are going to use a pre-trained model for the CIFAR10 set and we will see that with 50 steps it leaves a lot to be desired. In this case we are simply skipping 20 steps for each denoising we do, so that the image noise in one step does not match what the network estimate gives.

[36]:
```
model_id = "google/ddpm-cifar10-32"

ddpm = DDPMPipeline.from_pretrained(model_id).to(device)

nim = 5
samples = ddpm(batch_size=nim*nim,␣
 ↪num_inference_steps=inference_steps,output_type="numpy").images

fig = plt.figure(figsize=(5,5))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(samples[i])
    plt.axis('off')
plt.tight_layout(pad=0.)
```

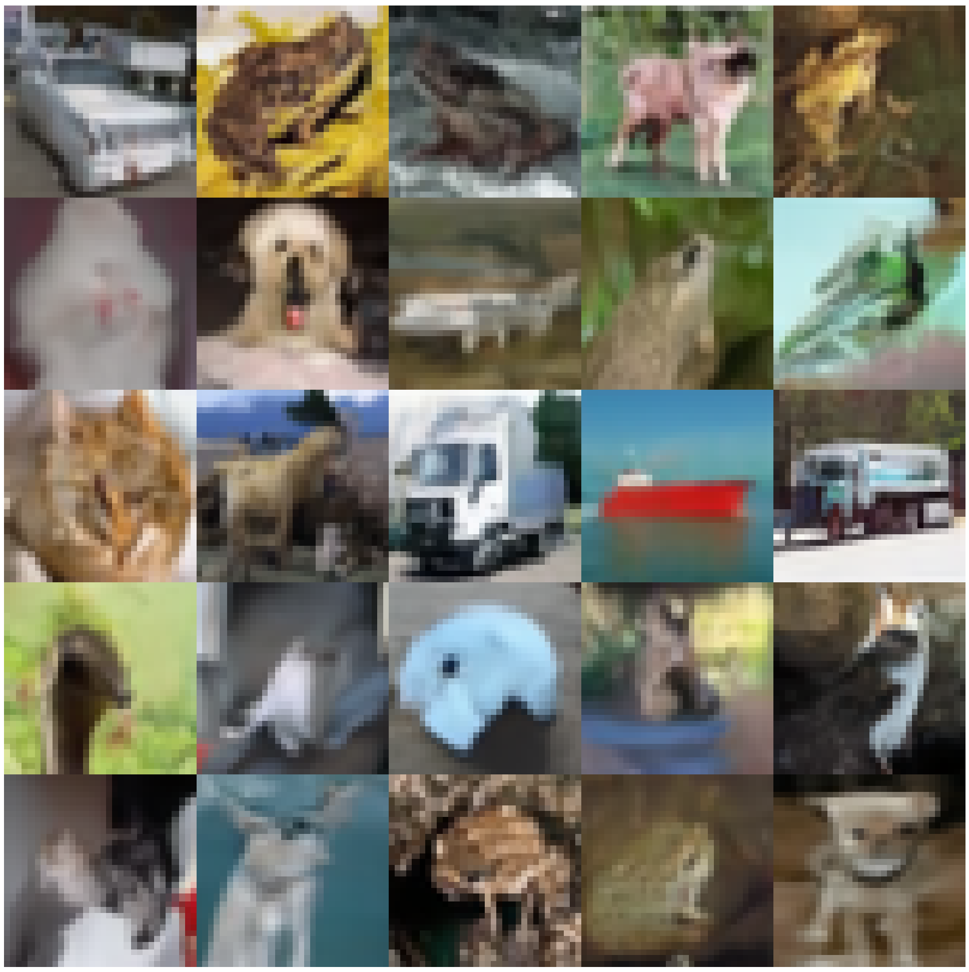


By skipping steps in DDIM the estimated noise is corrected so that it matches better (note that we are using the same noise estimation network trained with DDPM).

Obviously they are 32x32 images, but at least you can *imagine* what is in the image and there is no noise.

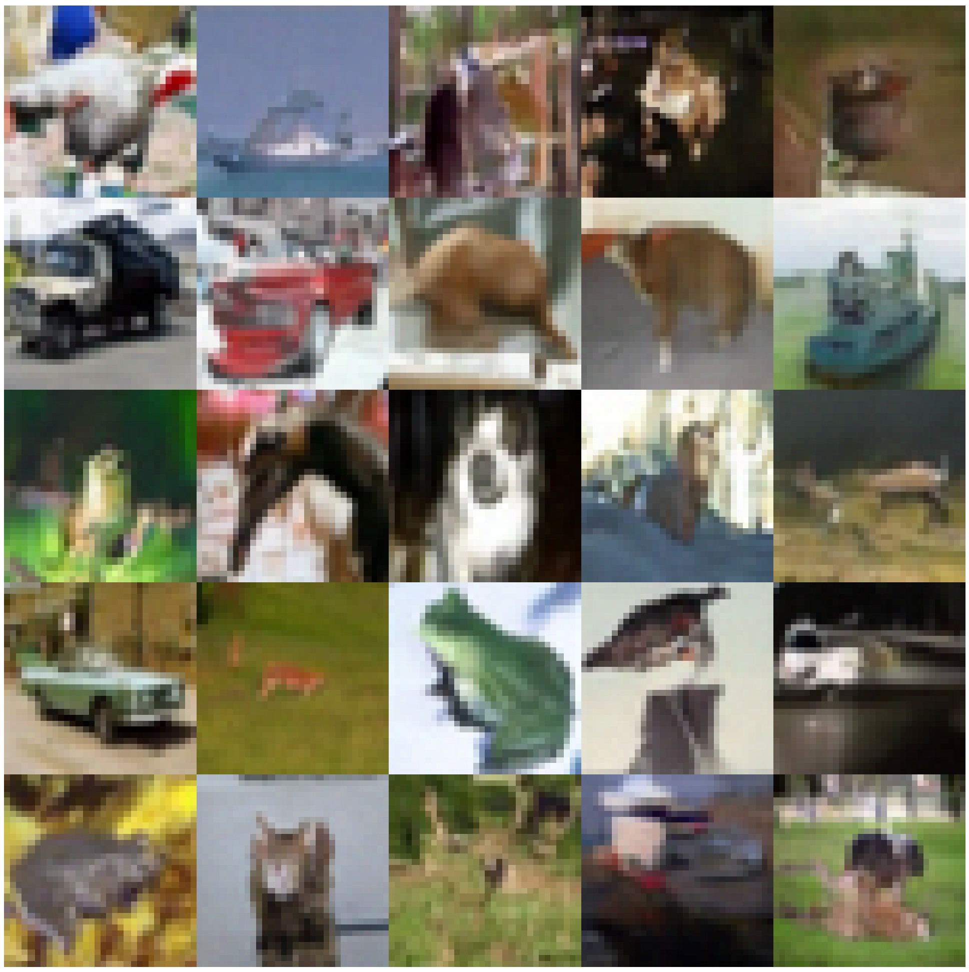
[37]:
```
ddim = DDIMPipeline.from_pretrained(model_id).to(device)
```

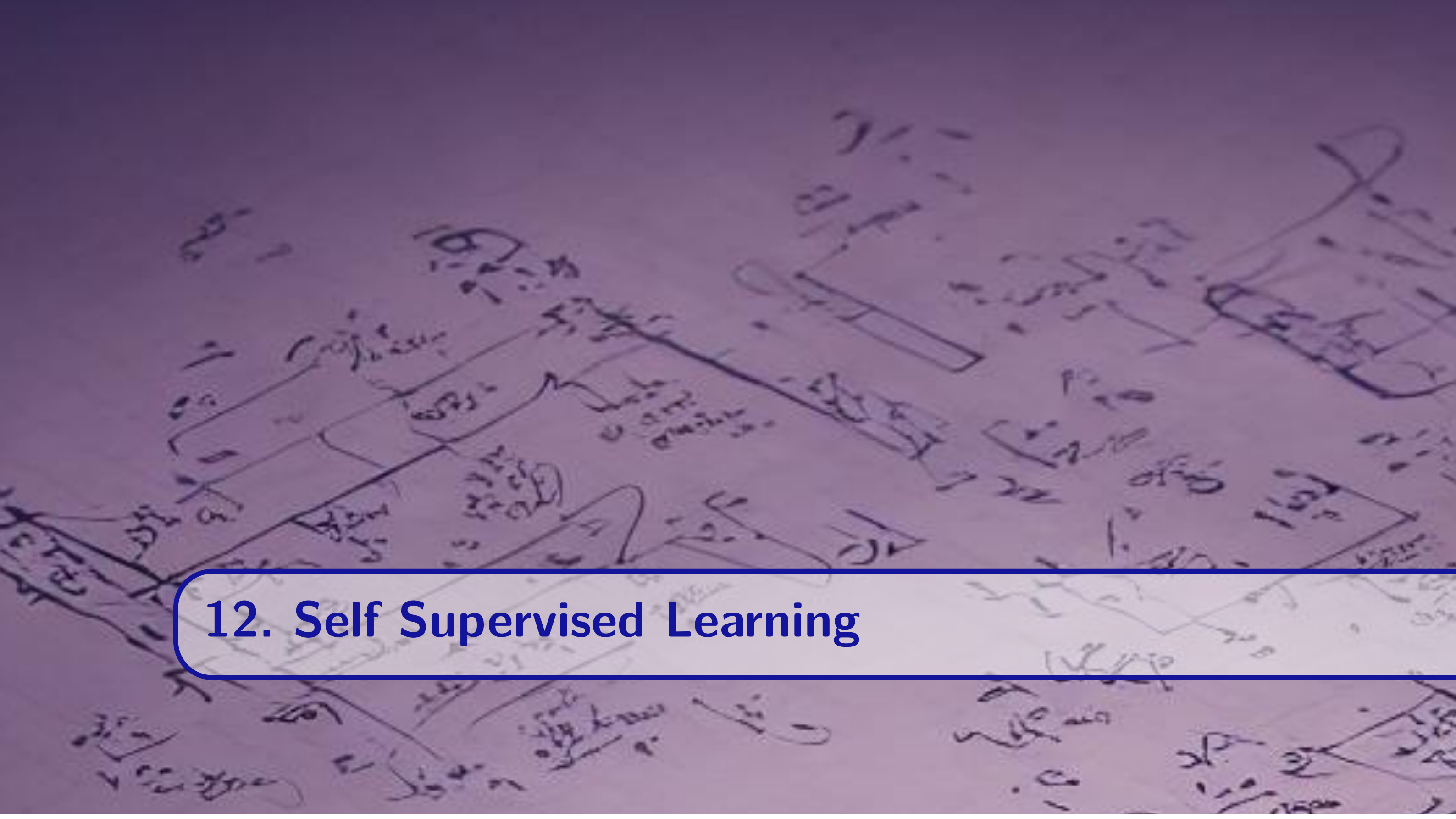
```
samples = ddim(batch_size=nim*nim,␣
 ↪num_inference_steps=inference_steps,output_type="numpy").images

fig = plt.figure(figsize=(5,5))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(samples[i])
    plt.axis('off')
plt.tight_layout(pad=0.)
```



Now we use the same noise prediction network but the scheduler solves it as a system of differential equations assuming that the trajectory between the noise and the image is continuous, the scheduler uses a resolution algorithm that allows it to adapt to the trajectory.

[38]:
```
pndm = PNDMPipeline.from_pretrained(model_id).to(device)

samples = pndm(batch_size=nim*nim,␣
 ↪num_inference_steps=inference_steps,output_type="numpy").images

fig = plt.figure(figsize=(5,5))
for i in range(nim*nim):
    plt.subplot(nim, nim, i+1)
    plt.imshow(samples[i])
    plt.axis('off')
plt.tight_layout(pad=0.)
```





12. Self Supervised Learning

12.1 Learning general representations

One of the problems when learning supervised tasks is that we need to collect a dataset and then produce the correct answer for all the data. This is certainly time-consuming, and it is not an easy task for many domains, where there are only a limited number of people with the expertise to solve it. It is easy to label cat photos, but you need a medical degree and years of experience to be able to detect and label a tumor in an MRI scan. Also, each problem/domain and type of task ends producing a different set of features that are hard to transfer to others.

It would be preferable to be able to learn general features that could be used for many tasks and domains without or minimal adjustment. In other words, we want to be able to learn some kind of universal representation without a specific final goal in mind. This is plausible, since we can see that any person is able to learn an internal representation for any task without constant supervision, so it seems a more natural way for learning. Also, we have and can collect many unsupervised data that we would be able to use.

It is obviously difficult to learn without any specific goal, but we can search for tasks that are hard enough that result in general representations when training a deep neural network to solve them. This is the aim of self supervised learning.

These representations will then help to solve supervised tasks reducing the amount of labeled data that is needed to adapt the pre-trained network to a new task (data efficiency) and obtaining a performance as good as using features specifically learned for the supervised task.

Self Supervised Learning is a sub area of deep unsupervised learning where the data provides its own supervision without any other source of information. This self supervision is a pretext task that is solved by a neural network. Roughly speaking, the kind of tasks that are useful for self supervised learning involve hiding part of the information of the data and predicting it using the remaining data.

The main difference between self supervised methods is what kind of pretext task or loss is used. If the problem to solve is hard enough, the network will be able to learn general features that can transferred to other tasks, specifically the supervised task we are usually interested in, like classification, regression, semantic segmentation, object localization...

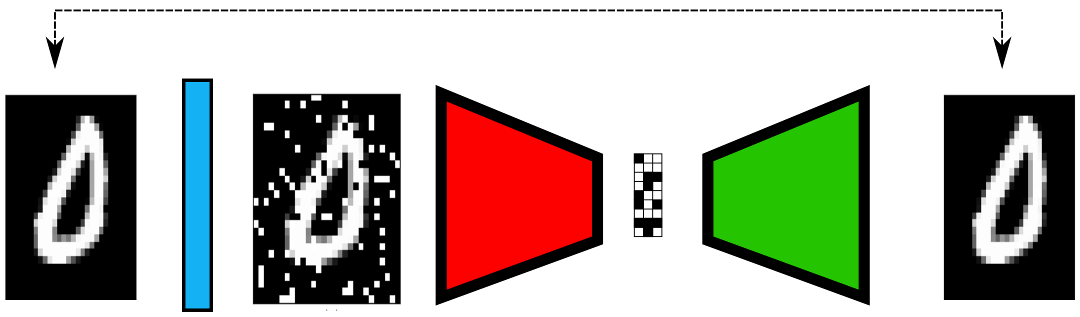


Figure 12.1: Denoising Autoencoder

The final goal of self supervised methods is to be able to learn better features (more general) than supervised methods that are transferable to many tasks with minimal cost.

Most of the current methods for self supervised learning are focused on vision, and we are going to explain mainly tasks that are related to images. Nevertheless, there are tasks that can be applied to any domain.

There are three main groups of tasks for performing self supervised learning that we will explain in the following sections:

- Reconstruction from partial or corrupted examples
- Common sense visual tasks easy to compute from a dataset with no labels
- Contrastive and non-contrastive tasks that aim to find a representation that separates examples that are different and puts together examples that are similar

12.2 Reconstruction

The reconstruction task is a fairly general one. It only involves destroying partially or hiding part of the example and obtaining a network that is able to predict the missing information from the context. This can be applied to images by occluding different parts or to any data that can be viewed as a sequence by hiding elements at any position.

12.2.1 Denoising

One of the firsts self supervised learning method of this kind pre-dates the deep learning era. Autoencoders as we explained on section 1.3.2 are neural networks that learn low dimensional representations of data by predicting an example from itself. Depending on how are trained, the representation could overfit. For preventing this, Denoising Autoencoders learn from corrupted examples. The examples are changed adding random noise before the input layer, and the network has to obtain the original examples (see figure 12.1). This helps to obtain more general features that do nor depend on specific elements of the input examples. The Denoising Autoencoder is trained with a loss that balances the error between the corrupted and original data using MSE or cross entropy depending on the type of example.

In [132] this network is used to learn a deep network layer by layer that can then be applied for supervised tasks. The paper as we have mentioned is previous to the deep learning era. At that time, it was not possible to train very large networks, let aside train networks with many layers.

The strategy for obtaining a deep network was to train a denoising autoencoder with one layer until convergence, then freeze the layer and use it to transform the data, this can

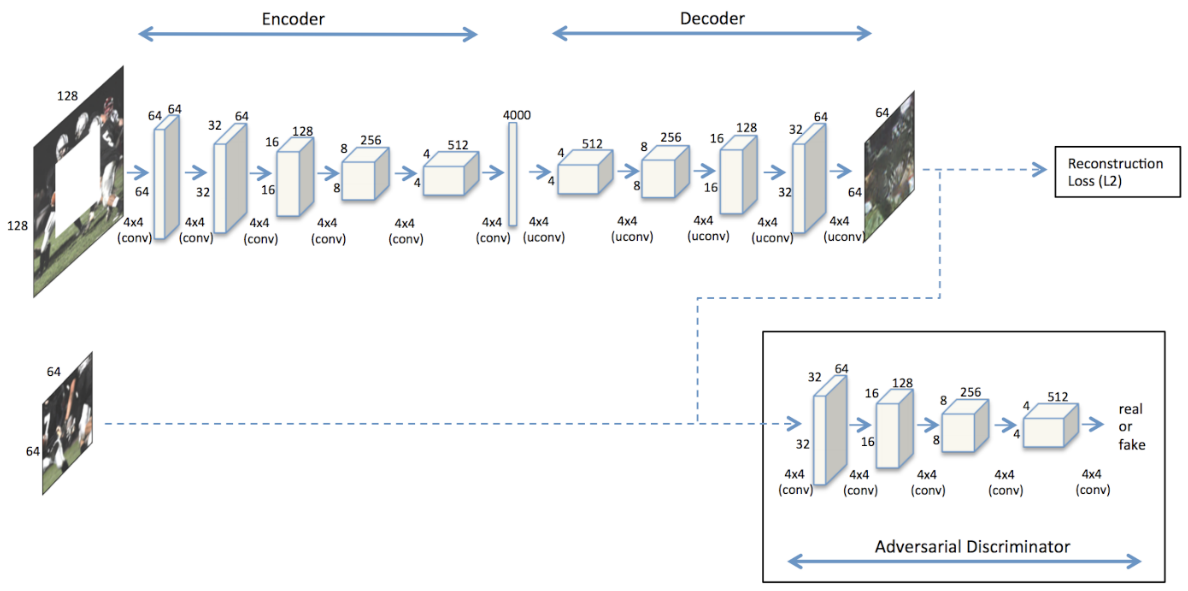


Figure 12.2: Learning representations with inpainting (source [108])

be precomputed and stored, so always a network with one layer is trained, reducing the computational cost. Then the next layer of the denoising autoencoder is trained. Now the corrupted data is the one transformed by the first trained layer, and the network can be fit to reproduce the transformed data. This can be repeated several times. Each layer has learned a general representation that does not rely on the whole output of the previous layer for predictions. Stacking all these layers we have a network that produces a general representation of the original input that can be fine-tuned to a supervised task, like for instance classification.

This was the beginning of self supervised learning, and the use of this technique was common before the generalized use of GPU training. This method was basically abandoned because GPUs made possible to train all the layers at the same time directly for the supervised task.

12.2.2 Inpainting

Back on the deep learning era, predicting for partial examples was a task that was already being solved using supervised learning. Specifically inpainting is a computer vision problem where the goal is to reconstruct an image that is incomplete. There is nothing that forbids to use this task for learning representation in a self supervised way, since we can transform any dataset into an inpainting dataset. We can just cut rectangular masks (or any other shape) in the images for producing a dataset. To solve this task the network will need some understanding of the context of the image, so it should be able to produce a fairly general representation.

This is the proposal of [108]. An encoder-decoder fully connected network that receives a masked image as input and outputs a proposal for the mask is paired with a discriminator network that determines of the image generated for the mask is real or fake, using adversarial training like in GANs (see figure 12.2). The encoder-decoder network receives combination of the reconstruction and adversarial losses. The idea is not to reproduce exactly the masked part of the image but to obtain inpaintings that have plausible content. After training the encoder can be then used for other tasks as feature extractor.

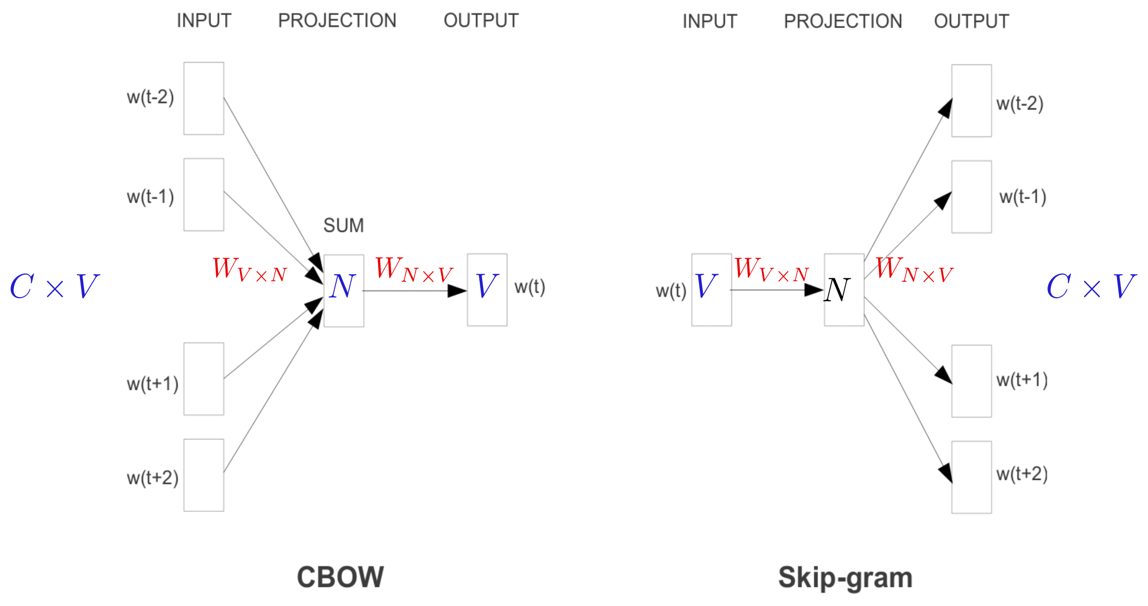


Figure 12.3: Word2vec architectures for learning representations for CBOW and skip grams

12.2.3 Masking

Word embeddings is a task that looks for good vector representations of words for natural language processing problems. Words have discrete representations, but it is more useful for deep learning methods to work with continuous representations. Before the deep learning era of natural language processing different methods have been proposed for transforming words into vector representation, like Singular Value Decomposition, Latent Dirichlet Allocation or Latent Semantic Analysis. The main drawback of these methods is their computational cost, the difficulty to cope with noisy datasets and with low frequency words. Deep learning has changed this using self supervised methods.

To find representations for word we can try to model their distribution using sentences. The probability of a sentence is a combination of the probability of its words and their co-ocurrences. We can use autoregressive distributions to represent the distribution of words assuming that the probability of a word depends on the surrounding words (bi-grams, tri-grams...) and their products will represent the probability of the sentence:

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=2}^{n} P(w_i | w_{i-1})$$

For learning these probabilities we only need a textual corpus without any additional task.

Word2Vec [93] proposes two models for learning word representations, the Continuous Bag of Words (CBOW), that generalizes the usual Bag of Words representation. This is based on predicting the probability distribution of a work given the surrounding words

$$P(w_i | w_{i-c}, \ldots, w_{i-1}, w_{i+1}, \ldots, w_{i+c})$$

and the Skip Gram representation, that is based on predicting the probability of the surrounding words given a word:

$$P(w_{i-c}, \ldots, w_{i-1}, w_{i+1}, \ldots, w_{i+c} | w_i)$$

The idea is basically to learn probabilities for a sequence using masking. It can be masking a word and predicting it from other words or masking all words except one and predicting them

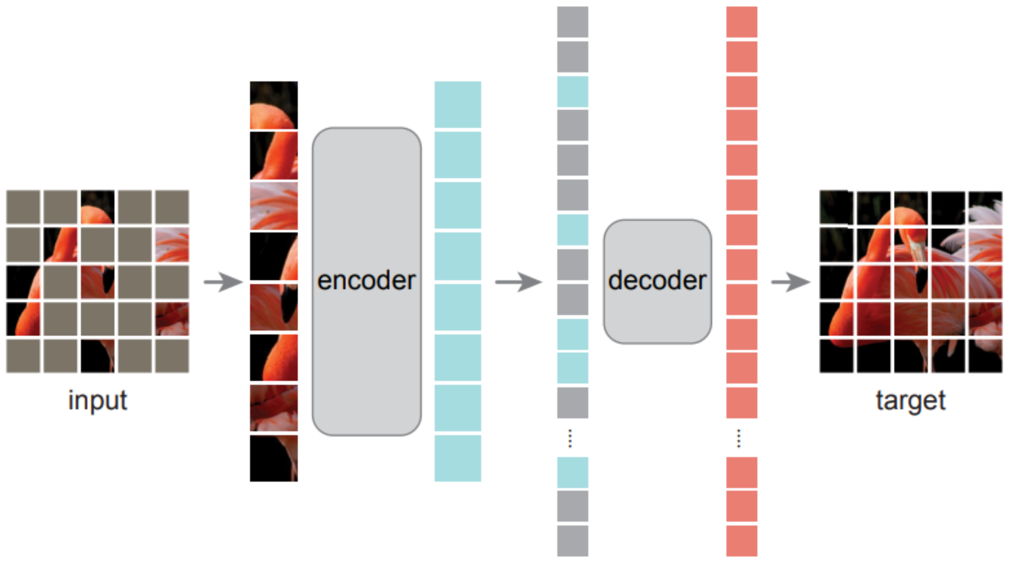


Figure 12.4: Masked Autoencoder learn an encoder predicting an image with a sequence of masked tokens (source 12.4)

from the word that is left. This can be done using neural networks that project one hot encoding representations of the words to a lower dimensionality space and computing the probabilities using the coordinates from the projection. Figure 12.3 shows the architectures for learing a representation for CBOW and skip-grams. The CBOW learns a matrix of weights that projects the words and then sums their representation for predicting the probability of the masked word and the skip-gram learns a matrix of weights that project a word and then uses it to predict the probabilities of all the masked words.

This was the beginning of a sequence of models that learn embeddings from text corpus using self supervised learning. The most advanced models like GPT and BERT are based on the Transformer architecture and are trained also using masking, where there is special text token that corresponds to the masked word that is predicted using the remaining tokens. The main change from the initial models is the use of transformers for the learning. This architecture allows also incorporating other tasks to the network with the addition of new tokens for the task that simplifies transferring the pretrained networks.

A similar idea has been proposed for images by the **Masked Auto Encoder** (MAE) [60]. For training this autoencoder images are cut on patches that are encoded using a network for obtaining tokensthat include positional encoding (this is similar to what is done in the vision transformer (ViT)).

A small percentage of these tokens (around 25%) is used to form a sequence where the missing tokens are substituted by a special token (mask token). A decoder predicts the missing tokens using the sequence similarly to the denoising autoencoder. The completed sequence of tokens is used to reconstruct the original image (pixels) and a loss that only considers the masked tokens is backpropagated to improve the encoding. After training the encoder is used for obtaining representations for other tasks. A representation of this process is in figure 12.4

12.3 Learning commonsense visual tasks

To reconstruct examples is not the only pretext task that can be used for self-supervised learning. The main constraint for a task is that it has to be hard, so the representation learned by the network also translates to other tasks. In this case the tasks that we are going to talk about are exclusively for vision, and they are difficult to adapt to other domains.

There are plenty of tasks that need general knowledge that we solve frequently. For instance, we are used to solving jigsaw puzzles. This is a challenging task that needs for the understanding of spatial relationships between the elements that are in the image. We can devise different tasks that use this as main element.

One possibility is presented by [35], where the task is to learn the position of image patches with respect to a reference patch. A central patch is extracted from an image and is paired with another one of its eight neighbors that are labeled with their positions (from 1 to 8). The network is given the two patches, for predicting the position of the neighbor. The patches are extracted so they do not overlap to avoid giving to the network information that can be used for an easy prediction. Also, some color transformations are applied to the patches and they are chosen so they are not aligned to avoid that defects on the image or other texture elements could make easier the task. Figure 12.7 show an example of this task.

A more complex task is described by [101]. In this case the network has to learn the permutation that corresponds to a set of nine patches selected also from a 3 by 3 grid with the constraint of not being aligned or to share border pixels. The processing network shares the parameters for all the patches and the output is passed through a classifier that predicts the number of the permutation used to order the patches. The permutations of the patches are chosen to maximize the hamming distance among them, so it is not trivial to compute the permutation.

Similarly, [51] proposes the task of predicting the rotation of an image. This is a classification task where the network is trained with full images using four possible rotations (no rotation, 90, 180, 270).

All these tasks are solved using networks that are commonly used for vision tasks like ResNet, ResNext, AlexNet or Inception. After the training different tasks can be solved using them as feature extractors, or fine-tuning them to the specific task, like classification, semantic segmentation or object localization.

12.4 Contrastive learning/non contrastive

A more general task that can work for different kinds of data is to learn to separate examples in the feature space. The idea is to find a representation that obtainsa similar output for similar examples that is also different from the representations from other examples. This can be taken to the extreme of learning representation that separates one example from the rest. This is related to the area of metric learning, where the goal is to learn distance metric function that corresponds to a subset of the distances or constraints among examples.

This can be approached from a probability estimation point of view, obtaining a measure that is able to distinguish samples that are from one distribution from examples out of distribution. For doing this, we can perform the estimation having a positive sample (the one from the distribution) and many negative samples (out of the distribution) using a **contrastive loss**.

This is the principle of the model Contrastive Predictive Coding (CPC) [103] that assumes that the data we want to obtain a representation from corresponds to a sequence. We can contrast elements at a point of the sequence to elements that are in their far future. This assures that there are some difference between the elements.

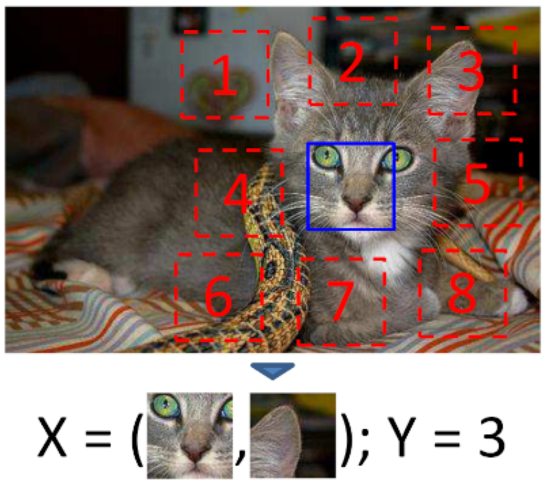


Figure 12.5: Self supervised learning by prediction the position of a patch of an image respect to a reference patch (source [35])

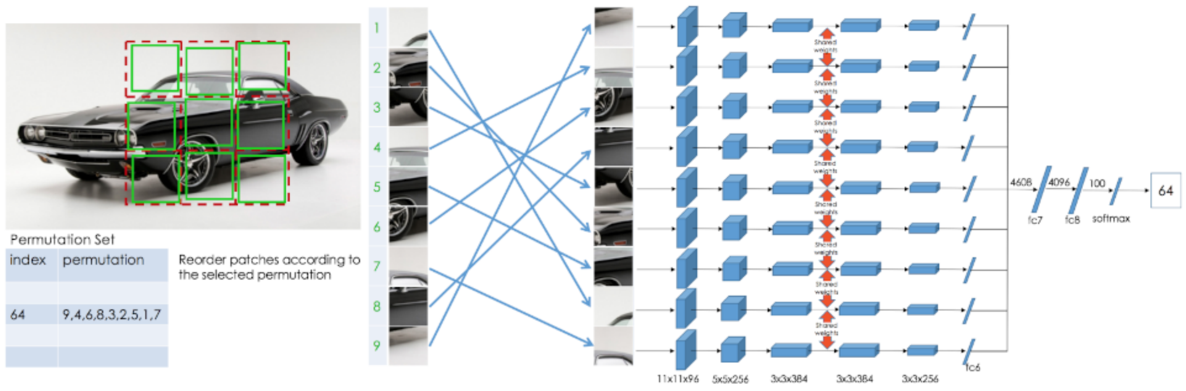


Figure 12.6: Self supervised learning by predicting the permutation that corresponds to a set of nine patches of an image (source [101])

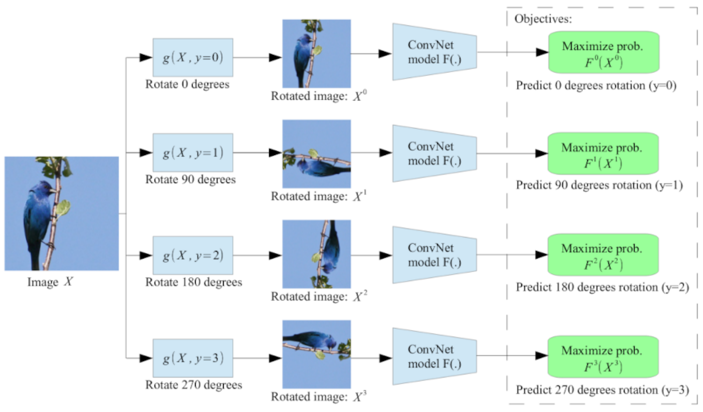


Image X
$g(X, y=0)$
Rotate 0 degrees
$g(X, y=1)$
Rotate 90 degrees
$g(X, y=2)$
Rotate 180 degrees
$g(X, y=3)$
Rotate 270 degrees
Rotated image: X^0
Rotated image: X^1
Rotated image: X^2
Rotated image: X^3
ConvNet model F(.)
Objectives:
Maximize prob. $F^0(X^0)$
Predict 0 degrees rotation (y=0)
Maximize prob. $F^1(X^1)$
Predict 90 degrees rotation (y=1)
Maximize prob. $F^2(X^2)$
Predict 180 degrees rotation (y=2)
Maximize prob. $F^3(X^3)$
Predict 270 degrees rotation (y=3)

Figure 12.7: Self supervised learning by predicting the rotation of an image (source [51])

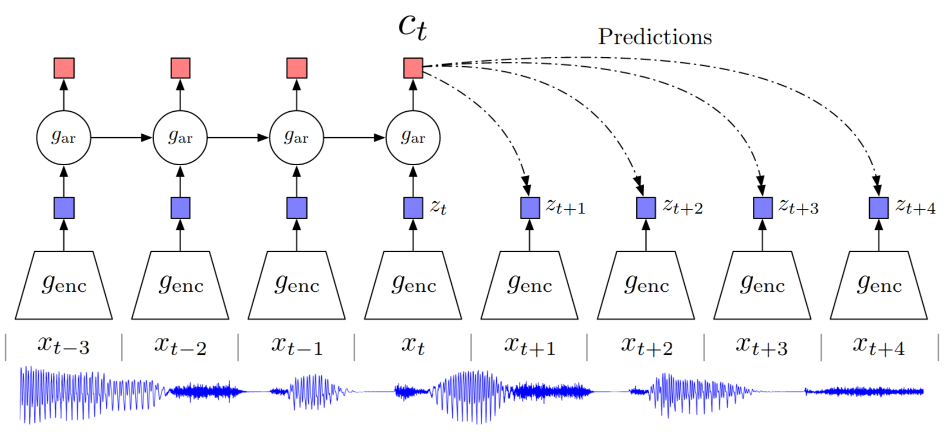


Figure 12.8: Contrastive learning on audio using CPC (source [103])

The model has an encoder network that applies a non-linear transformation to the examples to obtain a low dimensional representation $z_t = g_{enc}(x_t)$. We can obtain this representation for several examples close to a reference example (the positive one) and represent their relationships using an autoregressive distribution learned by another network $c_t = g_{ar}(z_{\leq t})$. This probability distribution can be used to make predictions of the probability of an example given a context. The goal is to approximate the ratio between the conditional distribution of an element x_{t+k} given its context $p(x_{t+k}|c_t)$, and an a priori distribution of the element $p(x_{t+k})$

$$f_k(x_{t+k}) \propto \frac{p(x_{t+k}|c_t)}{p(x_{t+k})}$$

This is a hard distribution to estimate directly, so CPC uses a noise contrastive loss named InfoNCE that uses one positive sample and a number of random negative samples that are chosen from the same sequence of the positive sample (x_{t+k}) at a distant future or from other sequences. The loss is

$$\mathcal{L}_N = -\mathbb{E}_X \left[\log \frac{f_k(x_{t+k}, c_t)}{\sum_{x_j \in X} f_k(x_j, c_t)} \right]$$

where we have N samples, one is the positive sample and $N-1$ are the negative ones. This loss will be better as the probability given to x_{t+k} by the context is higher than the one from the negative samples. The encoding network will learn a representation that makes this possible applying gradient descent over their parameters.

The paper applies this method to different domains where we have sequential information like audio, video and Natural Language Processing tasks. Figure 12.8 represents this model for learning representations for audio. The encoding network obtains representations for the chunks of audios that are used for learning an autoregressive distribution, this is used to contrast the probabilities of the predictions between positive and negative samples.

We can use other approaches that do not need to use sequences, and also define a hard problem that can help to obtain good representations. For instance to learn to obtain similar representation for different variations of the same example that is also different from other examples and their variations. This can be easily done for visual tasks using simply data augmentations. For images, we can compute augmentation and train an encoder network that obtains representations for the different augmentations that are similar and use a loss that encourages these representations to be far from the representations from other images. The more complex the augmentations, the harder will be the problem, and the better will be the

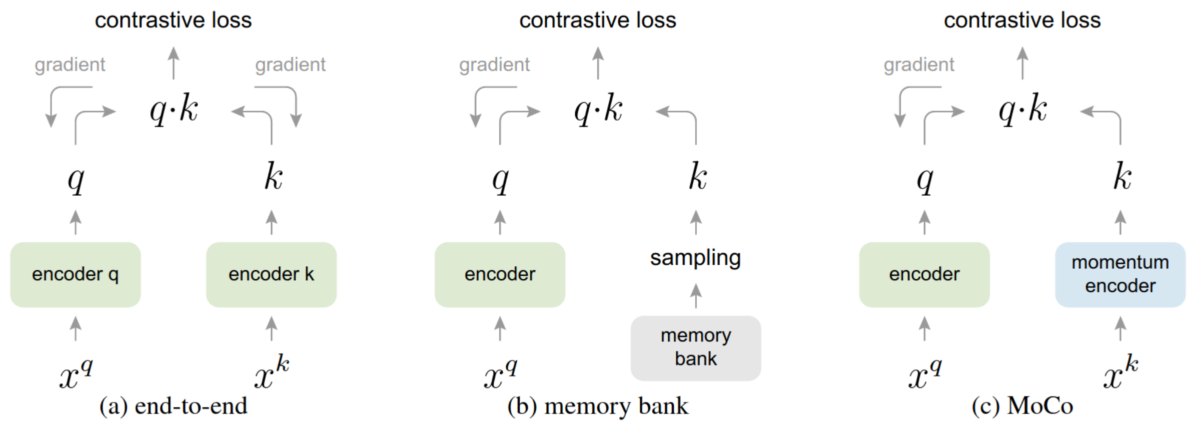


Figure 12.9: Training of Momentum Contrast model (source [59])

representation. This idea has generated a series of self supervised methods, we will review some of them.

Momentum Contrast [59] (MoCo) can be assimilated to the task of learning dynamic dictionaries. These dictionaries are learned with an encoder network that obtains a representation for images. This method is similar to siamese networks where a pair of networks are used to predict the distance between samples. In this case, we will have two identical networks that will be trained simultaneously to obtain similar representations, one is called the query network and the other the key network.

The model maintains a queue of representations that corresponds to the encoding of previous batches of examples. The negative samples will be taken from this structure. This allows obtaining a large number samples that does not depend on the batch size of the training. The elements in the dictionary are replaced as the training advances, deleting old representations and introducing the more recent ones. This is an alternative to other possible set-ups as having no dictionary and using the batch for obtaining the negative samples, with the disadvantage of limiting the number of possible negative samples or using only one network and maintaining a static dictionary with the problem of using outdated representations. Figure 12.9 represents all three variants.

During training, for each example in the batch two augmentations are obtained. One augmentation is encoded with the query network and the other with the key network, additionally a set of random encodings are taken from the dictionary for computing a contrastive loss with the dot product of the representations that corresponds to their similarity. The loss is a variation of the Noise Contrastive Estimation Loss:

$$\mathcal{L}(q,k^+,\{k^-\}) = -\log \frac{\exp(q \cdot k^+/\tau)}{\exp(q \cdot k^+/\tau) + \sum_{k^-} \exp(q \cdot k^-/\tau)}$$

where τ is a temperature hyperparameter that scales the values, so the result will be more or less sharp. Notice that this is basically a softmax, and scaling the exponential will make it more or less concentrated on a particular choice. This loss is computed for all examples on the training batch using the mean of the individual losses for the update of the networks.

This update is as follows (see also figure 12.9), the query network will be updated as usual applying the loss with backpropagation, the key network will update its parameters as a convex combination of the current weights and the weights of the query network $\theta_k = m\theta_k + (1-m)\theta_q$. This is equivalent to applying a momentum update to the parameters, hence the name of the model.

SimCLR [22] uses the end to end set-up that appears on figure 12.9, avoiding the use of

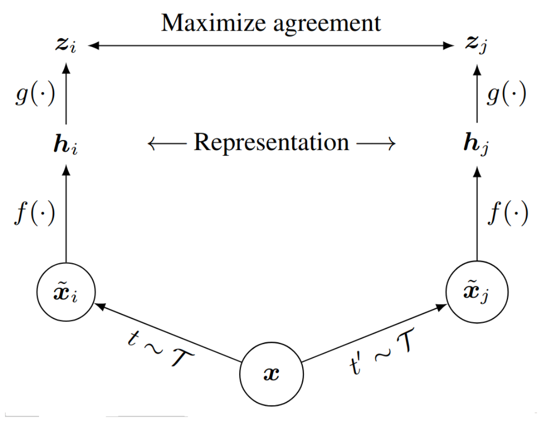


Figure 12.10: Training of Momentum Contrast model (source [22])

the dynamic dictionary and using only the examples in the batch as negative examples. The price of this is to have to use larger batches and longer training, but makes more simple the training. One key element of this method is to apply more than one augmentation to the positive sample making the problem harder as it results on better representations. Apart from the two encoder networks, an additional projection network (two layers MLP) is used to obtain a low dimensional vector that is used for computing the similarity of the representations. This is done to reduce the curse of dimensionality. This projection network is discarded after training. The results of the projection are used for computing a cosine similarity that is scaled to a specific range. Figure 12.10 represents the training procedure. This is used with the contrastive loss named Normalized Temperature-Scaled Cross Entropy loss (NT-Xent). All augmented examples are used to compute the loss increasing the negative sample size, so it is computed for $2N$ samples. The loss for a positive pair is

$$\mathcal{L}(i,j) = -\log \frac{exp(sim(z_i,z_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}[k \neq i] exp(sim(z_i,z_k)/\tau)}$$

The final loss is computed across all positive pairs and performed in both directions interchanging the key and query roles of the positive sample:

$$\mathcal{L} = \frac{1}{2N} \sum_{k=1}^{N} [\mathcal{L}(k,2k) + \mathcal{L}(2k,k)]$$

Given that the task of this method is to obtain similar representations for image augmentations, an alternative is to predict them instead of measuring their similarity. This opens the possibility of not using a contrastive loss for performing the task. This is the idea of the method Bootstrap your own latent (BYOL) [53] that also uses two twin networks, but adds a prediction network after one of them for predicting the projected representation of the second one. In this case MSE can be used as loss. This avoids the need of using negative samples, when computing the loss, but all the samples in a batch will have to be encoded just as in the other the methods. This means that we are only saving computation in the loss. Also, we are adding other network for predicting the projected representation.

Figure 12.11 represents the training procedure, two separate (but identical) networks encode two augmentations of an example in the batch, results are projected using an MLP and one of the projections is passed through a network that predict the other one. The loss is backpropagated through the branch that performs the prediction. The other network is updated using momentum as in the other methods.

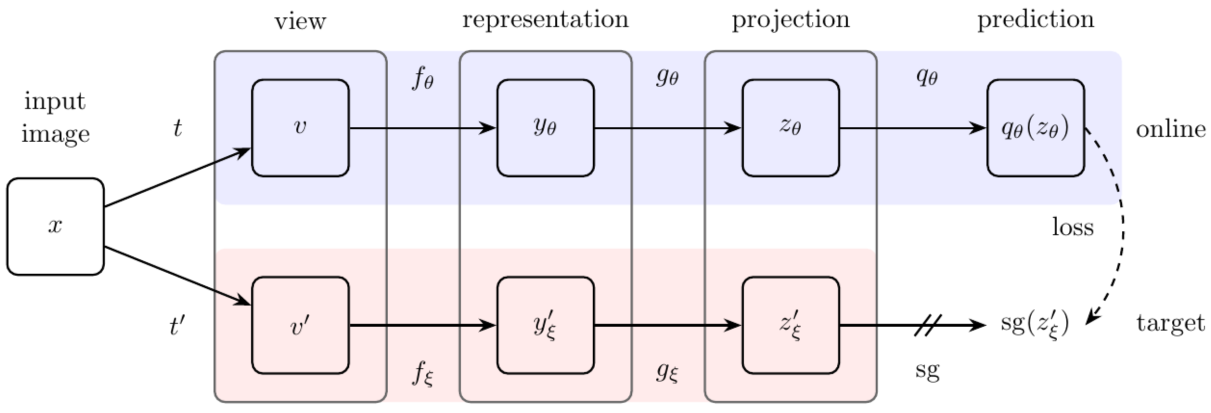


Figure 12.11: Training of BYOL model (source [53])

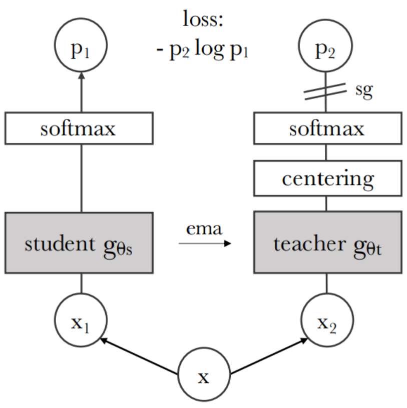


Figure 12.12: Training ofthe DINO model (source [19])

DINO [19] proposes a new alternative. This model is based on the Vision Transfomer (ViT) and works with image patches and attention. The learning is defined as a problem knowledge distillation where encoder network solves a classification problem. The classification is obtained by projecting the output of the encoder on k dimensions that correspond to artificial classes and defining their probabilities computing a sofmax.

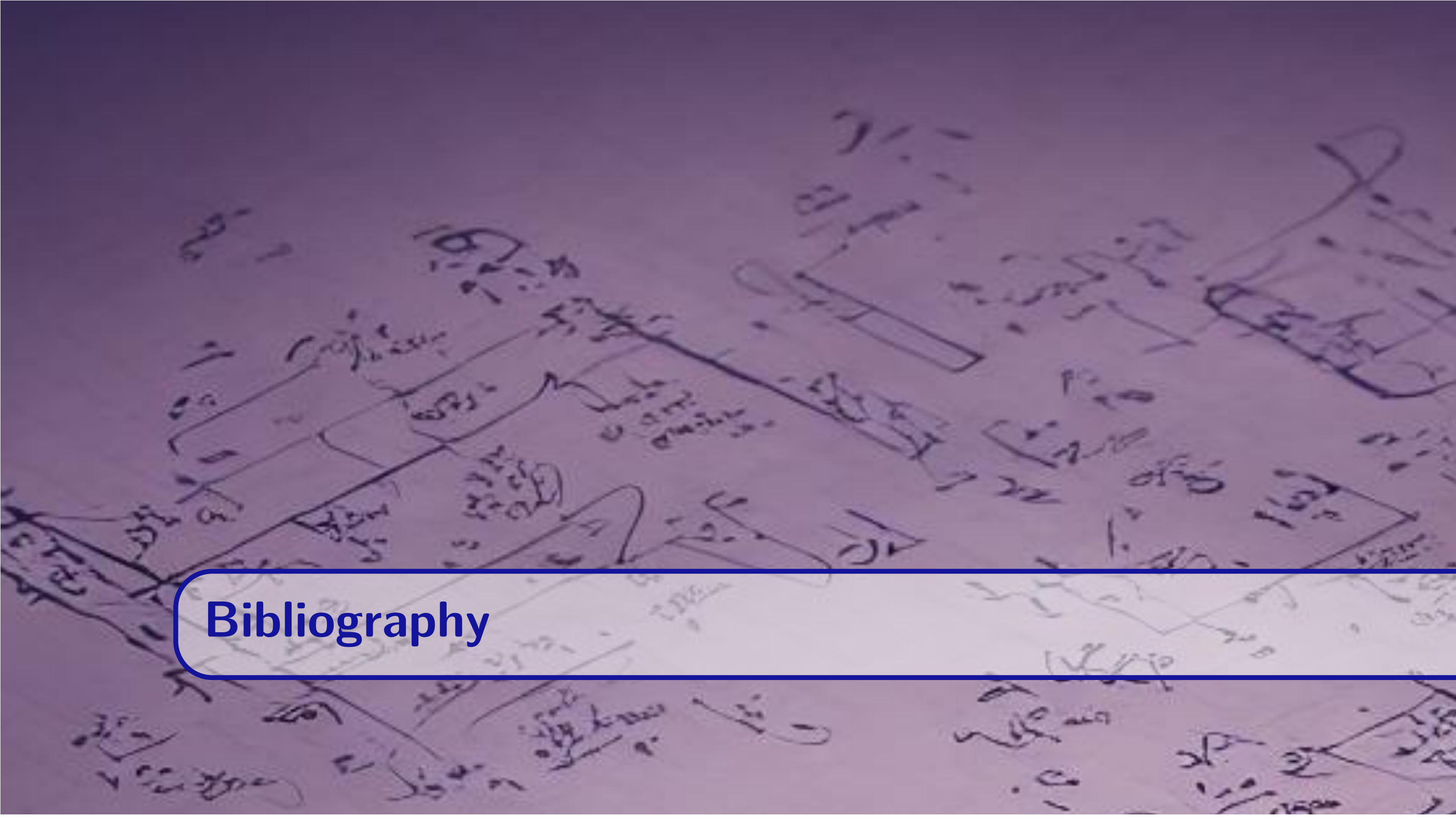
A student network has to reproduce the probabilities of the teacher. Each network receives image augmentations, but the teacher receives the complete image and the student a clipped image to make harder the problem. The loss function is cross entropy (see figure 12.12). As usual the student is updated with the loss and the teacher uses momentum. A property of this model is that learn in the attention maps to segment objects from the images, and is the basis of the Facebook's Segment Anything Model (SAM).

12.5 Evaluation

The goal final goal of these methods is to train any of the usual computer vision backbone networks as the encoder so it that can be applied later to different tasks. Being the training on a pretext task makes the evaluation of the results more complex. A simple approach is to use the features obtained by the trained network for learning a classifier. Usually a linear classifier (linear probe) is trained over the features using a test partition of the dataset used for training.

For instance, a logistic regression, linear SVM, k-nearest neighbor or a single layer perceptron can be trained to check if the classes are linearly separable with the obtained representation.

Since the goal is to obtain a representation agnostic for the task, also several computer vision tasks can be used for testing the representation solving for instance semantic segmentation or object localization benchmarks. This can be done by using directly the representation obtained from the pretrained network and training the other elements of the task from scratch or including a fine-tuning of the network. Classification tasks can also be solved, using more complex classifiers trained with other datasets, performing transfer learning with the frozen network or by fine-tuning also the network to adapt the representation for the new dataset.



Bibliography

[1] Charu C Aggarwal. “Outlier analysis”. In: *Data mining*. Springer. 2015, pages 237–263 (cited on page 15).

[2] Charu C. Aggarwal et al. “On Clustering Massive Data Streams: A Summarization Paradigm”. In: *Data Streams: Models and Algorithms*. Edited by Charu C. Aggarwal. Boston, MA: Springer US, 2007, pages 9–38. ISBN: 978-0-387-47534-9. DOI: 10.1007/978-0-387-47534-9_2 (cited on page 194).

[3] Rakesh Agrawal et al. “Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*. Volume 27,2. ACM SIGMOD Record. New York: ACM Press, June 1998, pages 94–105 (cited on page 124).

[4] Mihael Ankerst et al. “OPTICS: Ordering Points To Identify the Clustering Structure”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMod-99)*. Edited by Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh. Volume 28,2. SIGMOD Record. New York: ACM Press, June 1999, pages 49–60 (cited on page 121).

[5] Olatz Arbelaitz et al. “An extensive comparative study of cluster validity indices”. In: *Pattern Recognition* 46.1 (2013), pages 243–256. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2012.07.021. URL: http://www.sciencedirect.com/science/article/pii/S003132031200338X (cited on page 153).

[6] Martin Arjovsky, Soumith Chintala, and Léon Bottou. “Wasserstein generative adversarial networks”. In: *International conference on machine learning*. PMLR. 2017, pages 214–223 (cited on page 307).

[7] David Arthur and Sergei Vassilvitskii. “k-means++: the advantages of careful seeding”. In: *SODA*. Edited by Nikhil Bansal, Kirk Pruhs, and Clifford Stein. SIAM, 2007, pages 1027–1035. ISBN: 978-0-898716-24-5. URL: http://dl.acm.org/citation.cfm?id=1283383 (cited on page 97).

[8] Jacob Austin et al. “Structured denoising diffusion models in discrete state-spaces”. In: *arXiv preprint arXiv:2107.03006* (2021) (cited on page 359).

[9] Javad Azimi and Xiaoli Fern. "Adaptive cluster ensemble selection." In: *IJCAI*. Volume 9. 2009, pages 992–997 (cited on page 204).

[10] Adil M. Bagirov, Julien Ugon, and Dean Webb. "Fast modified global k-means algorithm for incremental cluster construction". In: *Pattern Recognition* 44.4 (2011), pages 866–876. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2010.10.018. URL: http://www.sciencedirect.com/science/article/pii/S0031320310005029 (cited on page 100).

[11] Fan Bao et al. "Analytic-dpm: an analytic estimate of the optimal reverse variance in diffusion probabilistic models". In: *arXiv preprint arXiv:2201.06503* (2022) (cited on page 350).

[12] Asa Ben-Hur et al. "Support Vector Clustering". In: *Journal of Machine Learning Research* 2 (2001), pages 125–137. URL: http://www.jmlr.org/papers/v2/horn01a.html (cited on page 137).

[13] Ella Bingham and Heikki Mannila. "Random projection in dimensionality reduction: applications to image and text data". In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2001, pages 245–250 (cited on page 46).

[14] Paul S. Bradley, Usama M. Fayyad, and Cory Reina. "Scaling Clustering Algorithms to Large Databases". In: *KDD*. Edited by Rakesh Agrawal, Paul E. Stolorz, and Gregory Piatetsky-Shapiro. AAAI Press, 1998, pages 9–15. ISBN: 1-57735-070-7 (cited on page 181).

[15] Andrew Brock, Jeff Donahue, and Karen Simonyan. "Large scale GAN training for high fidelity natural image synthesis". In: *arXiv preprint arXiv:1809.11096* (2018) (cited on pages 310–312).

[16] Christopher JC Burges. "Dimension reduction: A guided tour". In: *Foundations and Trends in Machine Learning* 2.4 (2009), pages 275–365 (cited on page 39).

[17] F. Camastra and A. Verri. "A Novel Kernel Method for Clustering". In: *IEEE Trans. Pattern Analysis and Machine Intelligence* 27.5 (May 2005), pages 801–804 (cited on page 138).

[18] Feng Cao et al. "Density-based clustering over an evolving data stream with noise". In: *Proceedings of the 2006 SIAM international conference on data mining*. SIAM. 2006, pages 328–339 (cited on page 194).

[19] Mathilde Caron et al. "Emerging properties in self-supervised vision transformers". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pages 9650–9660 (cited on page 385).

[20] Huiwen Chang et al. "Maskgit: Masked generative image transformer". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pages 11315–11325 (cited on page 360).

[21] Mark Chen et al. "Generative pretraining from pixels". In: *International conference on machine learning*. PMLR. 2020, pages 1691–1703 (cited on page 235).

[22] Ting Chen et al. "A simple framework for contrastive learning of visual representations". In: *International conference on machine learning*. PMLR. 2020, pages 1597–1607 (cited on pages 383, 384).

[23] Xi Chen et al. "Infogan: Interpretable representation learning by information maximizing generative adversarial nets". In: *arXiv preprint arXiv:1606.03657* (2016) (cited on page 318).

[24] Xi Chen et al. "Pixelsnail: An improved autoregressive generative model". In: *International Conference on Machine Learning*. PMLR. 2018, pages 864–872 (cited on pages 235, 236).

[25] Yixin Chen and Li Tu. "Density-based Clustering for Real-time Stream Data". In: *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '07. San Jose, California, USA: ACM, 2007, pages 133–142. ISBN: 978-1-59593-609-7. DOI: 10.1145/1281192.1281210 (cited on page 195).

[26] Sanjoy Dasgupta and Anupam Gupta. "An elementary proof of a theorem of Johnson and Lindenstrauss". In: *Random Structures & Algorithms* 22.1 (2003), pages 60–65 (cited on page 46).

[27] M. Dash et al. "Feature Selection for Clustering - A Filter Solution". In: *ICDM*. 2002, pages 115–122. ISBN: 0-7695-1754-4 (cited on page 38).

[28] Souptik Datta, Chris Giannella, and Hillol Kargupta. "Approximate distributed k-means clustering over a peer-to-peer network". In: *IEEE Transactions on Knowledge and Data Engineering* 21.10 (2009), pages 1372–1388 (cited on page 191).

[29] Michel M Deza and Elena Deza. *Encyclopedia of Distances*. Springer, 2013 (cited on page 67).

[30] Prafulla Dhariwal and Alex Nichol. *Diffusion Models Beat GANs on Image Synthesis*. 2021. arXiv: 2105.05233 [cs.LG] (cited on page 357).

[31] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. "Kernel k-means: spectral clustering and normalized cuts". In: *KDD*. Edited by Won Kim et al. ACM, 2004, pages 551–556. ISBN: 1-58113-888-1. URL: http://doi.acm.org/10.1145/1014052.1014118 (cited on page 100).

[32] Evgenia Dimitriadou, Andreas Weingessel, and Kurt Hornik. "Voting-Merging: An Ensemble Method for Clustering". In: *Lecture Notes in Computer Science* 2130 (2001), 217–?? ISSN: 0302-9743 (cited on page 205).

[33] Laurent Dinh, David Krueger, and Yoshua Bengio. "Nice: Non-linear independent components estimation". In: *arXiv preprint arXiv:1410.8516* (2014) (cited on page 255).

[34] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. "Density estimation using real nvp". In: *arXiv preprint arXiv:1605.08803* (2016) (cited on page 255).

[35] Carl Doersch, Abhinav Gupta, and Alexei A Efros. "Unsupervised visual representation learning by context prediction". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pages 1422–1430 (cited on pages 380, 381).

[36] Jeff Donahue, Philipp Krähenbühl, and Trevor Darrell. *Adversarial Feature Learning*. 2017. arXiv: 1605.09782 [cs.LG] (cited on page 319).

[37] Jeff Donahue and Karen Simonyan. *Large Scale Adversarial Representation Learning*. 2019. arXiv: 1907.02544 [cs.CV] (cited on page 319).

[38] David L Donoho and Carrie Grimes. "Hessian eigenmaps: Locally linear embedding techniques for high-dimensional data". In: *Proceedings of the National Academy of Sciences* 100.10 (2003), pages 5591–5596 (cited on page 45).

[39] R. Dubes and A Jain. *Algorithms for Clustering Data*. PHI Series in Computer Science. Prentice Hall, 1988 (cited on pages 80, 97).

[40] Conor Durkan et al. "Neural spline flows". In: *arXiv preprint arXiv:1906.04032* (2019) (cited on page 256).

[41] Patrick Esser et al. "Imagebart: Bidirectional context with multinomial diffusion for autoregressive image synthesis". In: *Advances in Neural Information Processing Systems* 34 (2021) (cited on page 360).

[42] Martin Ester et al. “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*. Edited by Evangelos Simoudis, Jia Wei Han, and Usama Fayyad. AAAI Press, 1996, page 226 (cited on page 119).

[43] Xiaoli Z. Fern and Carla E. Brodley. “Solving cluster ensemble problems by bipartite graph partitioning.” In: *ICML*. Edited by Carla E. Brodley. Volume 69. ACM International Conference Proceeding Series. ACM, Feb. 13, 2006 (cited on page 208).

[44] Xiaoli Z. Fern and Wei Lin. “Cluster Ensemble Selection”. In: *SDM*. SIAM, June 11, 2008, pages 787–797 (cited on page 204).

[45] Douglas H. Fisher. “Knowledge acquisition via incremental conceptual clustering”. In: *Machine learning* 2.2 (1987), pages 139–172 (cited on page 94).

[46] Ana L. N. Fred. “Finding Consistent Clusters in Data Partitions”. In: *Multiple Classifier Systems*. 2001, pages 309–318 (cited on page 206).

[47] Ana L. N. Fred and Anil K. Jain. “Combining Multiple Clusterings Using Evidence Accumulation”. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 27.6 (2005), pages 835–850 (cited on page 206).

[48] Frey and Dueck. “Clustering by Passing Messages Between Data Points”. In: *SCIENCE: Science* 315 (2007) (cited on page 135).

[49] John H Gennari, Pat Langley, and Doug Fisher. “Models of incremental concept formation”. In: *Artificial intelligence* 40.1 (1989), pages 11–61 (cited on page 93).

[50] Mathieu Germain et al. “Made: Masked autoencoder for distribution estimation”. In: *International Conference on Machine Learning*. PMLR. 2015, pages 881–889 (cited on pages 229, 230).

[51] Spyros Gidaris, Praveer Singh, and Nikos Komodakis. *Unsupervised Representation Learning by Predicting Image Rotations*. 2018. arXiv: 1803.07728 [cs.CV] (cited on pages 380, 381).

[52] Ian J Goodfellow et al. “Generative Adversarial Nets”. In: *stat* 1050 (2014), page 10 (cited on page 300).

[53] Jean-Bastien Grill et al. “Bootstrap your own latent: A new approach to self-supervised learning”. In: *arXiv preprint arXiv:2006.07733* (2020) (cited on pages 384, 385).

[54] S. Guha et al. “Clustering data streams: Theory and practice”. In: *Knowledge and Data Engineering, IEEE Transactions on* 15.3 (May 2003), pages 515–528. ISSN: 1041-4347. DOI: 10.1109/TKDE.2003.1198387 (cited on page 190).

[55] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. “CURE: An Efficient Clustering Algorithm for Large Databases”. In: *Inf. Syst* 26.1 (2001), pages 35–58. URL: http://dx.doi.org/10.1016/S0306-4379(01)00008-4 (cited on page 180).

[56] Ishaan Gulrajani et al. “Improved training of wasserstein gans”. In: *arXiv preprint arXiv:1704.00028* (2017) (cited on page 308).

[57] Stefan T. Hadjitodorov, Ludmila I. Kuncheva, and L. P. Todorova. “Moderate diversity for better cluster ensembles”. In: *Information Fusion* 7.3 (2006), pages 264–275 (cited on page 204).

[58] Trevor Hastie, Robert Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, July 2001 (cited on page 39).

[59] Kaiming He et al. “Momentum contrast for unsupervised visual representation learning”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pages 9729–9738 (cited on page 383).

[60] Kaiming He et al. “Masked autoencoders are scalable vision learners”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pages 16000–16009 (cited on page 379).

[61] X. He, D. Cai, and P. Niyogi. “Laplacian Score for Feature Selection”. In: *NIPS*. 2005 (cited on page 38).

[62] Martin Heusel et al. “Gans trained by a two time-scale update rule converge to a local nash equilibrium”. In: *arXiv preprint arXiv:1706.08500* (2017) (cited on page 302).

[63] Irina Higgins et al. “beta-vae: Learning basic visual concepts with a constrained variational framework”. In: (2016) (cited on page 280).

[64] Alexander Hinneburg and Daniel A Keim. “An efficient approach to clustering in large multimedia databases with noise”. In: *KDD*. Volume 98. 1998, pages 58–65 (cited on page 121).

[65] Alexander Hinneburg, Daniel A Keim, et al. “Optimal grid-clustering: Towards breaking the curse of dimensionality in high-dimensional clustering”. In: *VLDB*. Volume 99. Citeseer. 1999, pages 506–517 (cited on page 189).

[66] Jonathan Ho, Ajay Jain, and Pieter Abbeel. “Denoising diffusion probabilistic models”. In: *arXiv preprint arXiv:2006.11239* (2020) (cited on page 346).

[67] Jonathan Ho and Tim Salimans. “Classifier-Free Diffusion Guidance”. In: *NeurIPS 2021 Workshop on Deep Generative Models and Downstream Applications*. 2021 (cited on page 357).

[68] Victoria Hodge and Jim Austin. “A Survey of Outlier Detection Methodologies”. In: *Artificial Intelligence Review* 22.2 (2004). 10.1023/B:AIRE.0000045502.10941.a9, pages 85–126. ISSN: 0269-2821. URL: `http://dx.doi.org/10.1023/B:AIRE.0000045502.10941.a9` (cited on page 15).

[69] Emiel Hoogeboom, Rianne Van Den Berg, and Max Welling. “Emerging Convolutions for Generative Normalizing Flows”. In: *Proceedings of the 36th International Conference on Machine Learning*. Edited by Kamalika Chaudhuri and Ruslan Salakhutdinov. Volume 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pages 2771–2780. URL: `https://proceedings.mlr.press/v97/hoogeboom19a.html` (cited on page 256).

[70] Emiel Hoogeboom et al. “The convolution exponential and generalized sylvester flows”. In: *Advances in Neural Information Processing Systems* 33 (2020), pages 18249–18260 (cited on page 256).

[71] N. Iam-On et al. “A Link-Based Approach to the Cluster Ensemble Problem”. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 33.12 (Dec. 2011), pages 2396–2409. ISSN: 0162-8828. DOI: `10.1109/TPAMI.2011.84` (cited on page 206).

[72] Phillip Isola et al. “Image-to-image translation with conditional adversarial networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pages 1125–1134 (cited on pages 315, 317).

[73] T. Kanungo et al. “An efficient k-means clustering algorithm: analysis and implementation”. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 24.7 (July 2002), pages 881–892. ISSN: 0162-8828. DOI: `10.1109/TPAMI.2002.1017616` (cited on page 186).

[74] Mahdi Karami et al. “Invertible convolutional flow”. In: (2019) (cited on page 256).

[75] Tero Karras, Samuli Laine, and Timo Aila. “A style-based generator architecture for generative adversarial networks”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pages 4401–4410 (cited on pages 311, 313, 314).

[76] Tero Karras et al. “Progressive Growing of GANs for Improved Quality, Stability, and Variation”. In: *International Conference on Learning Representations*. 2018 (cited on page 308).

[77] Tero Karras et al. “Analyzing and improving the image quality of stylegan”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pages 8110–8119 (cited on pages 312, 315, 316).

[78] George Karypis, Eui-Hong Han, and Vipin Kumar. “Chameleon: Hierarchical clustering using dynamic modeling”. In: *Computer* 32.8 (1999), pages 68–75 (cited on page 131).

[79] Leonard Kaufman and Peter J Rousseeuw. “Partitioning around medoids (program pam)”. In: *Finding groups in data: an introduction to cluster analysis* (1990), pages 68–125 (cited on page 100).

[80] Diederik P Kingma and Prafulla Dhariwal. “Glow: Generative flow with invertible 1x1 convolutions”. In: *arXiv preprint arXiv:1807.03039* (2018) (cited on page 255).

[81] Diederik P. Kingma et al. *Variational Diffusion Models*. 2021. arXiv: 2107.00630 [cs.LG] (cited on page 350).

[82] R. Kohavi. “Wrappers for Feature Subset Selection”. In: *Art. Intel.* 97 (1997), pages 273–324 (cited on page 38).

[83] Tuomas Kynkäänniemi et al. “Improved precision and recall metric for assessing generative models”. In: *arXiv preprint arXiv:1904.06991* (2019) (cited on page 302).

[84] Tilman Lange et al. “Stability-based validation of clustering solutions”. In: *Neural computation* 16.6 (2004), pages 1299–1323 (cited on page 160).

[85] Ping Li, Trevor J Hastie, and Kenneth W Church. “Very sparse random projections”. In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2006, pages 287–296 (cited on page 46).

[86] Tao Li, Chris H. Q. Ding, and Michael I. Jordan. “Solving Consensus and Semi-supervised Clustering Problems Using Nonnegative Matrix Factorization”. In: *ICDM*. IEEE Computer Society, 2007, pages 577–582. URL: http://doi.ieeecomputersociety.org/10.1109/ICDM.2007.98 (cited on page 211).

[87] A. C. Likas, N. Vlassis, and J. J. Verbeek. “The global k-means clustering algorithm”. In: *Pattern Recognition* 36.2 (Feb. 2003), pages 451–461. URL: http://www.sciencedirect.com/science/article/B6V14-45TTJY0-7/2/1bfdf4def0cf8b6ba77fd25132df8d0d (cited on page 99).

[88] Ulrike Luxburg. “A tutorial on spectral clustering”. English. In: *Statistics and Computing* 17.4 (2007), pages 395–416. ISSN: 0960-3174. DOI: 10.1007/s11222-007-9033-z. URL: http://dx.doi.org/10.1007/s11222-007-9033-z (cited on page 132).

[89] Laurens van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE”. In: *Journal of machine learning research* 9.Nov (2008), pages 2579–2605 (cited on page 47).

[90] Hoda Mashayekhi et al. “GDCluster: a general decentralized clustering algorithm”. In: *IEEE transactions on knowledge and data engineering* 27.7 (2015), pages 1892–1905 (cited on page 192).

[91] A. McCallum, K. Nigam, and L. Ungar. “Efficient clustering of high-dimensional data sets with application to reference matching”. In: *KDD* (2000), pages 169–178 (cited on page 184).

[92] Chenlin Meng et al. “Sdedit: Image synthesis and editing with stochastic differential equations”. In: *arXiv preprint arXiv:2108.01073* (2021) (cited on page 358).

[93] Tomas Mikolov et al. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013) (cited on page 378).

[94] Mehdi Mirza and Simon Osindero. “Conditional generative adversarial nets”. In: *arXiv preprint arXiv:1411.1784* (2014) (cited on page 315).

[95] Takeru Miyato and Masanori Koyama. "cGANs with projection discriminator". In: *arXiv preprint arXiv:1802.05637* (2018) (cited on pages 315, 316).

[96] Takeru Miyato et al. "Spectral normalization for generative adversarial networks". In: *arXiv preprint arXiv:1802.05957* (2018) (cited on page 309).

[97] G Bel Mufti, P Bertrand, and LE Moubarki. "Determining the number of groups from measures of cluster stability". In: *Proceedings of international symposium on applied stochastic models and data analysis*. 2005, pages 17–20 (cited on page 160).

[98] Muhammad Ferjad Naeem et al. "Reliable fidelity and diversity metrics for generative models". In: *International Conference on Machine Learning*. PMLR. 2020, pages 7176–7185 (cited on page 302).

[99] Murilo Coelho Naldi, ACPLF Carvalho, and Ricardo JGB Campello. "Cluster ensemble selection based on relative validity indexes". In: *Data Mining and Knowledge Discovery* 27.2 (2013), pages 259–289 (cited on page 204).

[100] Alex Nichol and Prafulla Dhariwal. *Improved Denoising Diffusion Probabilistic Models*. 2021. arXiv: `2102.09672 [cs.LG]` (cited on page 350).

[101] Mehdi Noroozi and Paolo Favaro. "Unsupervised learning of visual representations by solving jigsaw puzzles". In: *European conference on computer vision*. Springer. 2016, pages 69–84 (cited on pages 380, 381).

[102] Augustus Odena, Christopher Olah, and Jonathon Shlens. "Conditional image synthesis with auxiliary classifier gans". In: *International conference on machine learning*. PMLR. 2017, pages 2642–2651 (cited on page 315).

[103] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. *Representation Learning with Contrastive Predictive Coding*. 2019. arXiv: `1807.03748 [cs.LG]` (cited on pages 380, 382).

[104] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. *Neural Discrete Representation Learning*. 2018. arXiv: `1711.00937 [cs.LG]` (cited on pages 280, 281).

[105] Aaron Oord et al. "Parallel wavenet: Fast high-fidelity speech synthesis". In: *International conference on machine learning*. PMLR. 2018, pages 3918–3926 (cited on page 254).

[106] Aaron van den Oord et al. "Conditional image generation with pixelcnn decoders". In: *arXiv preprint arXiv:1606.05328* (2016) (cited on pages 232, 233).

[107] Aaron van den Oord et al. "Wavenet: A generative model for raw audio". In: *arXiv preprint arXiv:1609.03499* (2016) (cited on page 231).

[108] Deepak Pathak et al. "Context Encoders: Feature Learning by Inpainting". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016 (cited on page 377).

[109] Bidyut Kr. Patra, Sukumar Nandi, and P. Viswanath. "A distance based clustering method for arbitrary shaped clusters in large datasets". In: *Pattern Recognition* 44.12 (2011), pages 2862–2870. URL: `http://dx.doi.org/10.1016/j.patcog.2011.04.027` (cited on page 180).

[110] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks". In: *arXiv preprint arXiv:1511.06434* (2015) (cited on pages 304–306).

[111] Ali Razavi, Aaron van den Oord, and Oriol Vinyals. "Generating diverse high-fidelity images with vq-vae-2". In: *arXiv preprint arXiv:1906.00446* (2019) (cited on pages 280, 282).

[112] Scott Reed et al. "Generative adversarial text to image synthesis". In: *International conference on machine learning*. PMLR. 2016, pages 1060–1069 (cited on page 315).

[113] S. T. Roweis and L. K. Saul. "Nonlinear Dimensionality Reduction by Locally Linear Embedding". In: *Science* 290.5500 (Dec. 2000), pages 2323–2326. URL: `http://www.sciencemag.org/content/vol290/issue5500/` (cited on page 45).

[114] Chitwan Saharia et al. *Image Super-Resolution via Iterative Refinement*. 2021. arXiv: `2104.07636 [eess.IV]` (cited on page 358).

[115] Chitwan Saharia et al. "Palette: Image-to-image diffusion models". In: *arXiv preprint arXiv:2111.05826* (2021) (cited on page 358).

[116] Mehdi SM Sajjadi et al. "Assessing generative models via precision and recall". In: *arXiv preprint arXiv:1806.00035* (2018) (cited on page 302).

[117] Tim Salimans and Jonathan Ho. "Progressive distillation for fast sampling of diffusion models". In: *arXiv preprint arXiv:2202.00512* (2022) (cited on page 355).

[118] Tim Salimans et al. "Improved techniques for training gans". In: *arXiv preprint arXiv:1606.03498* (2016) (cited on page 301).

[119] Tim Salimans et al. "Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications". In: *arXiv preprint arXiv:1701.05517* (2017) (cited on pages 233, 234).

[120] Sergio M Savaresi and Daniel L Boley. "A comparative analysis on the bisecting K-means and the PDDP clustering algorithms". In: *Intelligent Data Analysis* 8.4 (2004), pages 345–362 (cited on page 99).

[121] D. Sculley. "Web-scale k-means clustering". In: *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pages 1177–1178 (cited on page 184).

[122] Jiaming Song, Chenlin Meng, and Stefano Ermon. "Denoising diffusion implicit models". In: *arXiv preprint arXiv:2010.02502* (2020) (cited on page 354).

[123] Yang Song and Stefano Ermon. "Generative modeling by estimating gradients of the data distribution". In: *arXiv preprint arXiv:1907.05600* (2019) (cited on page 351).

[124] Yang Song and Stefano Ermon. "Improved techniques for training score-based generative models". In: *arXiv preprint arXiv:2006.09011* (2020) (cited on page 351).

[125] Yang Song et al. "Score-based generative modeling through stochastic differential equations". In: *arXiv preprint arXiv:2011.13456* (2020) (cited on page 352).

[126] A. Strehl and Joydeep Ghosh. "Cluster ensembles- A knowledge reuse framework for combining multiple partitions." In: *Journal of Machine Learning Research* 3.3 (2003), pages 583–617 (cited on page 207).

[127] R. Tibshirani, W. Guenther, and T. Hastie. "Estimating the number of clusters in a dataset via the gap statistic". In: *Journal of the Royal Statistical Society B* (2000) (cited on page 159).

[128] Alexander P. Topchy, Anil K. Jain, and William F. Punch. "Clustering Ensembles: Models of Consensus and Weak Partitions". In: *IEEE Trans. Pattern Analysis and Machine Intelligence* 27.12 (Dec. 2005), pages 1866–1881. URL: `http://dx.doi.org/10.1109/TPAMI.2005.237` (cited on pages 209, 210).

[129] Arash Vahdat, Karsten Kreis, and Jan Kautz. "Score-based generative modeling in latent space". In: *Advances in Neural Information Processing Systems* 34 () (cited on page 356).

[130] Aaron Van Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. "Pixel recurrent neural networks". In: *International Conference on Machine Learning*. PMLR. 2016, pages 1747–1756 (cited on pages 231, 232).

[131] Ashish Vaswani et al. "Attention is all you need". In: *arXiv preprint arXiv:1706.03762* (2017) (cited on pages 234, 235, 310).

[132] Pascal Vincent et al. "Extracting and composing robust features with denoising autoencoders". In: *Proceedings of the 25th international conference on Machine learning*. 2008, pages 1096–1103 (cited on page 376).

[133] P. Viswanath and V. Suresh Babu. "Rough-DBSCAN: A fast hybrid density based clustering method for large data sets". In: *Pattern Recognition Letters* 30.16 (2009), pages 1477–1488. ISSN: 0167-8655. DOI: 10.1016/j.patrec.2009.08.008. URL: http://www.sciencedirect.com/science/article/pii/S0167865509002153 (cited on page 188).

[134] Wei Wang, Jiong Yang, and Richard R. Muntz. "STING: A Statistical Information Grid Approach to Spatial Data Mining". In: *Twenty-Third International Conference on Very Large Data Bases*. Edited by Matthias Jarke et al. Athens, Greece: Morgan Kaufmann, 1997, pages 186–195 (cited on page 123).

[135] Y. Wang and Y. Zhang. "Non-negative Matrix Factorization: a Comprehensive Review". In: *Knowledge and Data Engineering, IEEE Transactions on* 25.6 (2013), pages 1336–1353 (cited on page 44).

[136] Martin Wattenberg, Fernanda Viégas, and Ian Johnson. "How to use t-sne effectively". In: *Distill* 1.10 (2016), e2 (cited on page 47).

[137] L. Wolf and A. Shashua. "Feature Selection for Unsupervised and Supervised Inference". In: *Journal of Machine Learning Research* 6 (2005), pages 1855–1887 (cited on page 38).

[138] Zhisheng Xiao, Karsten Kreis, and Arash Vahdat. "Tackling the generative learning trilemma with denoising diffusion gans". In: *arXiv preprint arXiv:2112.07804* (2021) (cited on page 355).

[139] Zhiwen Yu and Hau-San Wong. "Quantization-based clustering algorithm". In: *Pattern Recognition* 43.8 (2010), pages 2698–2711. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2010.02.020. URL: http://www.sciencedirect.com/science/article/pii/S0031320310000981 (cited on page 187).

[140] H. Zeng and Yiu-ming Cheung. "A new feature selection method for Gaussian mixture clustering". In: *Pattern Recognition* 42.2 (Feb. 2009), pages 243–250 (cited on page 38).

[141] Han Zhang et al. "Self-attention generative adversarial networks". In: *International conference on machine learning*. PMLR. 2019, pages 7354–7363 (cited on page 310).

[142] Kai Zhang, Ivor W. Tsang, and James T. Kwok. "Maximum Margin Clustering Made Practical". In: *IEEE Transactions on Neural Networks* 20.4 (2009), pages 583–596. URL: http://dx.doi.org/10.1109/TNN.2008.2010620 (cited on page 138).

[143] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. "BIRCH: A New Data Clustering Algorithm and Its Applications". In: *Data Min. Knowl. Discov* 1.2 (1997), pages 141–182. URL: http://dx.doi.org/10.1023/A:1009783824328 (cited on page 182).

[144] Zhen-yue Zhang and Hong-yuan Zha. "Principal manifolds and nonlinear dimensionality reduction via tangent space alignment". In: *Journal of Shanghai University (English Edition)* 8.4 (2004), pages 406–424 (cited on page 45).

[145] Weizhong Zhao, Huifang Ma, and Qing He. "Parallel k-means clustering based on mapreduce". In: *IEEE International Conference on Cloud Computing*. Springer. 2009, pages 674–679 (cited on page 190).

[146] Jun-Yan Zhu et al. "Unpaired image-to-image translation using cycle-consistent adversarial networks". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pages 2223–2232 (cited on pages 316, 318).