Unsupervised Machine Learning and Data Mining

(Lecture Notes)

Javier Béjar

AMLT

Master in Artificial Intelligence

(Draft October 17, 2016)
# Contents

1 Data Preprocessing 3  
1.1 Datasets 3  
1.2 Data Preprocessing 4  
1.2.1 Outliers 4  
1.2.2 Missing Values 5  
1.2.3 Data normalization 6  
1.2.4 Data discretization 7  
1.3 Dimensionality reduction 8  
1.3.1 Feature selection 9  
1.3.2 Feature extraction 10  
1.4 Similarity/distance functions 18  
1.5 Examples 20  
1.5.1 Dimensionality reduction and discrete data 20  
1.5.2 Application: Wheelchair control 21  

2 Clustering 25  
2.1 Statistical Hierarchical clustering 26  
2.1.1 Graph based algorithms 26  
2.1.2 Matrix algebra based algorithms 27  
2.1.3 Drawbacks 29  
2.2 Conceptual Clustering 29  
2.3 Partitional clustering Algorithms 32  
2.3.1 Prototype/model based clustering 33  
2.3.2 Density based clustering 42  
2.3.3 Grid based clustering 45  
2.3.4 Other approaches 47  
2.4 Applications 54  
2.4.1 Twitter Geoprofiling 54  

3 Clustering Validation 59  
3.1 Measures for assessing clusterness 60  
3.2 Measures for clustering validation 60  
3.2.1 Internal validity criteria 60  
3.2.2 External criteria 62  
3.2.3 Relative Criteria 64
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>Assessing the number of clusters</td>
<td>65</td>
</tr>
<tr>
<td>3.4</td>
<td>Other methods for clustering validation</td>
<td>67</td>
</tr>
<tr>
<td>3.5</td>
<td>Applications</td>
<td>68</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Wheelchair control</td>
<td>68</td>
</tr>
<tr>
<td>4</td>
<td>Clustering Large Datasets</td>
<td>71</td>
</tr>
<tr>
<td>4.1</td>
<td>Scalability strategies</td>
<td>71</td>
</tr>
<tr>
<td>4.1.1</td>
<td>One-pass strategies</td>
<td>72</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Summarization Strategies</td>
<td>72</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Sampling/batch strategies</td>
<td>73</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Approximation strategies</td>
<td>73</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Divide and conquer strategies</td>
<td>73</td>
</tr>
<tr>
<td>4.2</td>
<td>Algorithms</td>
<td>74</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Scalable hierarchical clustering</td>
<td>74</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Rough-DBSCAN</td>
<td>74</td>
</tr>
<tr>
<td>4.2.3</td>
<td>CURE</td>
<td>75</td>
</tr>
<tr>
<td>4.2.4</td>
<td>BIRCH</td>
<td>75</td>
</tr>
<tr>
<td>4.2.5</td>
<td>OptiGrid</td>
<td>77</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Scalable K-means</td>
<td>78</td>
</tr>
<tr>
<td>4.2.7</td>
<td>STREAM LSEARCH</td>
<td>79</td>
</tr>
<tr>
<td>4.2.8</td>
<td>Mini batch K-means</td>
<td>80</td>
</tr>
<tr>
<td>4.2.9</td>
<td>Canopy clustering</td>
<td>81</td>
</tr>
<tr>
<td>4.2.10</td>
<td>Indexed K-means</td>
<td>82</td>
</tr>
<tr>
<td>4.2.11</td>
<td>Quantized K-means</td>
<td>84</td>
</tr>
<tr>
<td>5</td>
<td>Semisupervised Clustering</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>Biasing using labeled examples</td>
<td>87</td>
</tr>
<tr>
<td>5.2</td>
<td>Biasing using instance constraints</td>
<td>88</td>
</tr>
<tr>
<td>5.3</td>
<td>Distance Learning</td>
<td>90</td>
</tr>
<tr>
<td>6</td>
<td>Association Rules</td>
<td>93</td>
</tr>
<tr>
<td>6.1</td>
<td>Definitions</td>
<td>94</td>
</tr>
<tr>
<td>6.2</td>
<td>The Apriori algorithm</td>
<td>95</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Example</td>
<td>98</td>
</tr>
<tr>
<td>6.3</td>
<td>Mining without candidate generation</td>
<td>101</td>
</tr>
<tr>
<td>6.4</td>
<td>Measure for association rules interestingness</td>
<td>106</td>
</tr>
<tr>
<td>6.5</td>
<td>Applications</td>
<td>107</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Discovering geographical associations from Twitter Data</td>
<td>107</td>
</tr>
<tr>
<td>7</td>
<td>Structured Datasets</td>
<td>111</td>
</tr>
<tr>
<td>7.1</td>
<td>Mining sequential data</td>
<td>111</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Clustering for temporal series</td>
<td>112</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Frequent sequences</td>
<td>113</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Clustering data streams</td>
<td>119</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Application: Sequential data</td>
<td>120</td>
</tr>
<tr>
<td>7.2</td>
<td>Graph mining</td>
<td>122</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Graph partitioning</td>
<td>122</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Clustering of graphs/trees</td>
<td>126</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Frequent subgraphs</td>
<td>126</td>
</tr>
</tbody>
</table>
These are the lecture notes for the first part of the course Advanced Machine Learning Techniques (AMLT). The idea of these notes is to facilitate following 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 programmed for the class.

The different chapters have examples using several machine learning libraries mainly developed in python. It is important that you have installed these libraries in your computer so you can run the different examples and make your own experiments.

Most of the examples use the scikit-learn library, that is available for any python distribution using pip or easy_install. Some methods are implemented in the library AMLT-learn that you can find in https://github.com/bejar/AMLT-learn. This library can also be installed using pip (pip install AMLT-learn)
1.1 Datasets

Each domain has its own representation according to its characteristics. 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 by a flat structure, we will call them unstructured datasets. These datasets 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 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 (time related or with a domain specific order relationship) and tree/graph structures. In a sequence we have an order (total or partial) relationship among all the examples of the dataset. This order usually corresponds with time (time series) or a domain defined precedence like nucleotids order in DNA. This sequence can contain itself unstructured examples, being a sequence of unstructured data, for instance, text obtained from a news feed 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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.1</td>
<td>green</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5.7</td>
<td>blue</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-2.2</td>
<td>blue</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-9.0</td>
<td>red</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.3</td>
<td>yellow</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2.1</td>
<td>green</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.1: Example of the representation of an unstructured dataset
CHAPTER 1. DATA PREPROCESSING

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

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 (e.g. 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 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 rest of the values of the dataset. These examples can be considered
as having erroneous 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.

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

There are different algorithms for detecting outliers in data that can be divided into parametric and non-parametric methods.

Parametric methods assume a specific probability distribution for the attributes, they use the data to estimate its parameters, and then they compute the probability of the values. Those that have a probability lower than a specified threshold are marked as outliers. Other possibility is to use the upper and lower quartiles of the distribution and mark as outliers all the values that are at a distance larger than a threshold from the quartiles.

Sometimes it is not possible to assume a model for the distribution of the values, so a non parametric method has to be used. There are different possibilities, for example, to compute a histogram for the attributes (deciding an adequate binning) and to discard the examples that are from bins that have a low number of examples. Other possible solution is to apply kernel density estimation (KDE) to approximate locally the probability distribution and to label as outliers the examples that belong to low density areas.

All these methods detect attributes that have outlier values. An example with only a few outliers has still valuable information, so a threshold has to be determined to decide when the example is different enough to be considered an outlier respect to the rest of the data.

Another non parametric methods use a global approach (not focused on only a single attribute) using a proximity criteria to determine if the example itself is an outlier by. They use the distribution of the distances to the k-nearest neighbors of each example. If this distribution is different enough from the mean behavior, then the example can be labeled as an outlier and discarded.

An excellent review on this topic can be found in [44].

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 \( O(|\text{examples}|) \) 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 \( O(|\text{examples}| \times \log(k)) \), 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 first method will not affect the mean of the data. The rest will have less impact in the variance. Figure 1.4 represents the result of missing values imputation using the mean and the 1-nn.

1.2.3 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**

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 \([0,1]\)
\[ \text{range}_\text{norm}(x) = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \]

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 \((\mu_x, \sigma_x)\) so all the attributes have \(\mu = 0\) and \(\sigma = 1\). The following transformation performs this change:

\[ Z_{\text{norm}}(x) = \frac{x - \mu_x}{\sigma_x} \]

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:

\[ s_x = \frac{1}{N} \sum_{i=0}^{N} |x - \mu_x| \]

### 1.2.4 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 1.5. 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 1.6 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.

### 1.3 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, that 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 instances. 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 it is the case, and sometimes the specific goal of the mining process is to find particular groups of instances 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. DIMENSIONALITY REDUCTION

1.3.1 Feature selection

Most of the research on feature selection is related to supervised learning [49]. More recently, methods for unsupervised learning have appeared in the literature [23], [41], [74], [78]. 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 more easy 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 ([41]) 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 ($\sigma$), so the weights of the edges are:

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

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 is 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:

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

The score $L_r$ is then computed as:

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

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 [18].

**Principal Component Analysis (PCA)**

The most popular method for feature extraction is Principal Component Analysis [40]. 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
1.3. DIMENSIONALITY REDUCTION

Figure 1.7: PCA computes a transformation that needs less dimensionality to explain the variance of the data
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 are enough for its representation.

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 1.7 represents the idea 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:

\[ f(\lambda) = \mu + V_q \lambda \]

\( \mu \) is a location vector in \( \mathbb{R}^p \), \( V_q \) is a \( p \times q \) matrix of \( q \) orthogonal unit vectors and \( \lambda \) is a \( q \) vector of parameters. We want to minimize the reconstruction error for the data:

\[ \min_{\mu, \{\lambda_i\}, V_q} \sum_{i=1}^{N} ||x_i - \mu - V_q \lambda_i||^2 \]

Optimizing partially for \( \mu \) and \( \lambda_i \) (taking partial derivatives) we have:

\[ \mu = \bar{x} \]

\[ \lambda_i = V_{iq}^T(x_i - \bar{x}) \]

We can obtain the matrix \( V_q \) by minimizing:

\[ \min_{V_q} \sum_{i=0}^{N} \|(x_i - \bar{x}) - V_q V_q^T(x_i - \bar{x})\|_2 \]
Assuming $\bar{x} = 0$ we can obtain the projection matrix $H_q = V_q V_q^T$ by Singular Value Decomposition of the data matrix $X$

$$X = UDV^T$$

$U$ is a $N \times p$ orthogonal matrix, its columns are the left singular vectors $V$ 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 component to keep to obtain a specific amount of explained variance.

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

**PCA variations**

PCA is a linear transformation, 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. This method is known as **Kernel PCA**. Distances among the examples are defined as a dot product that can be obtained using a kernel:

$$d(x_i, x_j) = \Phi(x_i)^T \Phi(x_j) = K(x_i, x_j)$$

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:

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

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 $\ell^1$ 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:

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

The $\ell^1$ norm regularization will encourage sparse solutions (zero eigenvalues).

Figure 1.8 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 1.6. 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 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 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 (distorsion) that leads to several variants, such as least squares MDS, Sammon 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:

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

The optimization problem is defined as:

\[ \arg \min_{z_1, z_2, ..., z_n} S_D(z_1, z_2, ..., z_n) \]

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 using the derivative of \( S_D(z_1, z_2, ..., z_n) \) and a step \( \alpha \) 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:

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

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

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

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:

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

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 uses a similar idea than PCA, an eigen decomposition of a matrix that transforms the original data in the product of two matrices. The main difference is that the values of the matrices are constrained to be positive. The formulation assumes that the data is the sum of unknown positive latent variables and the positiveness assumption helps to the interpretation of the result. 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 [73].

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 1.9 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
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
Local Linear Embedding

Local Linear Embedding ([61, 27, 81]) performs a transformation that preserves the local structure of the data. It assumes that each instance can be reconstructed by a linear combination of its neighbors. The method computes for each example a set of reconstruction weights from its k-nearest neighbors. From these weights a new set of coordinates in a lower dimensional space is computed for each data point. Different variants of the algorithm have been defined to address the problem of using more neighbors than dimensions in a transformation. This circumstance can sometimes result in a space that does not represent the original space.

The general algorithm for this method is as follows:

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

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

   \[
   \min_{W_{ik}} \| x_i - \sum_{k \in N(i)} w_{ik}x_k \|^2
   \]

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

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

   \[
   \sum_{i=0}^{N} \| y_i - \sum_{k=0}^{N} w_{ik}y_k \|^2
   \]

   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 \( N \) where a pair \((i, i')\) belongs to this set if \( i \) is among the \( k \) neighbors of \( i' \) or viceversa, the objective function that is minimized is:
1.3. DIMENSIONALITY REDUCTION

Figure 1.11: ISOMAP, LLE and Random Projection transformation of the iris dataset

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

The parameter $\tau$ 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 ([16], [52]) 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:

  \[
  \begin{align*}
  \frac{s}{\sqrt{n\_components}} & \quad \text{with probability} \quad 1 - \frac{1}{s} \\
  0 & \quad \text{with probability} \quad \frac{1}{2s} \\
  -\frac{s}{\sqrt{n\_components}} & \quad \text{with probability} \quad \frac{1}{2s}
  \end{align*}
  \]

  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 ([22]) can be used to obtain an estimate of the number of dimensions from the number of examples in the dataset and the expected distortion.
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 \emph{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**
  \[
  d(i, k) = \frac{x_i^T \cdot x_j}{\|x_i\| \cdot \|x_j\|}
  \]
  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**
  \[
  d(i, k) = \frac{(x_i - \bar{x}_i)^T \cdot (x_j - \bar{x}_j)}{\|x_i - \bar{x}_i\| \cdot \|x_j - \bar{x}_j\|}
  \]
  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 \emph{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**
  \[
  d(i, k) = \left( \sum_{j=1}^{d} |x_{ij} - x_{kj}|^r \right)^{\frac{1}{r}}
  \]
  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.
1.4. SIMILARITY/DISTANCE FUNCTIONS

Figure 1.12: Distance/similarity matrices for the Iris dataset using Euclidean, Cosine and Mahalanobis functions

- **Mahalanobis distance**

\[ d(i, k) = (x_i - x_k)^T \cdot \varphi^{-1} \cdot (x_i - x_k) \]

where \( \varphi \) 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**

\[ d(i, k) = \max_j |x_{ij} - x_{kj}| \]

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

- **Canberra distance**

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

This is a weighted variation of the Manhattan distance.

Figure 1.12 shows the distance/similarity matrix for the Iris dataset (among all the examples) computed for three functions (euclidean, cosine and mahalanobis). The chosen function changes the relations among examples, resulting in a different space embedding, so they have to be chosen carefully so the patterns in the data are correctly identified.

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 [24].
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. 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.

1.5.2 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 in 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 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 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 nonlinear transformations reveal a similar structure. In this case LLE and ISOMAP have been computed with a large number of 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 hiding larger clusters.
Figure 1.14: Mushroom data visualization after Random Projection, PCA and ISOMAP (100 and 5 neighbors) transformation
1.5. EXAMPLES

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)
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. 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 in the patterns that show the values of the attributes that describe the dataset. The goal of the task will be to either obtain a representation that describes a 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 a $N$-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 have to be compact (dense) groups of examples that are maximally separated from other groups in the dataset. Broadly, 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 matrix of examples distances/similarities and a recursive divisive or agglomerative strategy.

- **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 discovery strategy used. The more representative ones include algorithms based on prototypes or probabilistical 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.

2.1 Statistical Hierarchical clustering

Hierarchical methods [28] use two strategies for building a tree of nested clusters that partitions 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.

2.1.1 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.

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

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
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

---

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 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 happens 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.

### 2.1.2 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
Algorithm 2.2 Algorithm for matrix algebra based agglomerative hierarchical clustering

**Algorithm:** Agglomerative Clustering

1. Compute Distance/similarity matrix
2. 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
3. until Matrix has one element

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 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 continuous of algorithms that allows obtaining very different partitions from the same data. Most of the usual criteria can be described using the following formula:

\[
d((i,j), k) = \alpha_i d(i, k) + \alpha_j d(j, k) + \beta d(i, j) + \gamma [d(i, k) - d(j, k)]
\]

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:

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>2,3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>2,3</td>
<td>7.5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now the closest groups are the examples 1 and 4, and after the distance recomputation:
And finally the example 5 is the closest to the group (1,4).

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.

### 2.1.3 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 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 problems 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 criteria 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.

### 2.2 Conceptual Clustering

Related to hierarchical clustering but from the perspective of unsupervised machine learning, conceptual clustering ([35]) 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
Figure 2.1: Dendrograms obtained applying different linkage criteria to the same dataset

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.
2.2. CONCEPTUAL CLUSTERING

The COBWEB algorithm ([33]) 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 an 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:

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

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

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

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):

\[
\sum_{i=1}^{I} \sum_{j=1}^{J} P(A_i = V_{ij})^2
\]

The category utility measure compares the difference between having a specific partition of the data and no having any partition at all in the following way:

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

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:

\[
CU(\{C_1, \ldots C_K\}) = \frac{\sum_{k=1}^{K} P(C_k) \sum_{i=1}^{I} \frac{1}{\sigma_{ik}} - \sum_{i=1}^{I} \frac{1}{\sigma_{ip}}}{K}
\]

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)
2.3 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,
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 $\text{CU}$ of incorporating the example to each class
| Save the two best $\text{CU}$
| Compute $\text{CU}$ of merging the best two classes
| Compute $\text{CU}$ of splitting the best class
| Compute $\text{CU}$ of creating a new class with the example
|
| Recursive call with the best choice

end

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

2.3.1 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 [28]. 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:

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

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 a priori 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
Algorithm 2.4 K-means algorithm

**Algorithm:** K-means (X: Examples, k: integer)

1. Generate k initial prototypes (e.g. the first k examples)
2. Assign the k examples to its nearest prototype
3. SumD = Sum of square distances examples-prototypes

```
repeat
    Recalculate prototypes
    Reassign examples to its nearest prototype
    SumI = SumD
    SumD = Sum of square distances examples-prototypes
until SumI - SumD < ϵ
```

Figure 2.4: K-means algorithm example

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 centers. As the clusters are well separated each iteration the new centers are closer to the cluster centroids, converging in just tree steps of the algorithm in this case to the true clusters.

This algorithm has some drawbacks, being an approximate algorithm, there is no guarantee about the quality of the solution. It converges to a local minima 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++ ([9]) proposes a seeding technique based on randomization that has proven to yield better results. The idea is to try to maximize the distance among initial centers so they cover most of the
Figure 2.5: Clusters with different sizes and densities are problematic for K-means, left actual data labels, right partition obtained by k-means

space of examples. The algorithm is as follows:

1. Choose one center 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 center that has already been chosen

3. Choose one new data point at random as a new center, 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 \( k \) centers 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 larger cluster into the smallest 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 of the problems the original algorithm has or that include other methodologies that allow the discovery of other than spherical clusters. The first one that is worth mentioning is Bisecting K-means ([63]). 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 criteria 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 ([53]) 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 [12] 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 ([25]) 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 has to change, optimizing the sum of distances
from each example to the medoid of their cluster. This changes the method 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) ([47]), 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 medoid $m$
   - For each non-medoid $o$: Swap $m$ 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 incre-
mental 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 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.

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
than at a radius apart from any other cluster. This means that in any given dimension cannot
be more than $\left(\frac{l}{2r} - 1\right)$ 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.
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 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**

**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 covariances 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:

\[
P(x|\theta) = \sum_{i=1}^{K} P(w_i) P(x|\theta_i, w_i)
\]

With \( K \) the number of clusters, and \( \sum_{i=1}^{K} P(w_i) = 1 \). 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:

$$P(x|\mu, \Sigma) = \sum_{i=1}^{K} P(w_i) P(x|\mu_i, \Sigma_i, w_i)$$
The actual computations depend on the assumptions that we make about the attributes (independent or not, same \( \sigma_i \)). If the attributes are assumed independent \( \mu_i \) and \( \sigma_i \) have to be computed for each class \( (O(k) \) parameters) (the model is described by hyper spheres or ellipsoids parallel to coordinate axis). If the attributes are modeled as not independent \( \mu_i \), \( \sigma_i \) and \( \sigma_{ij} \) have to be computed for each class \( (O(k^2) \) parameters) (the model is described by hyper ellipsoids non parallel to coordinate axis).

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

\[
P(x | \vec{\mu}_i, \Sigma_i, w_i) = \prod_{j=1}^{A} P(x | \mu_{ij}, \sigma_{ij}, w_i)
\]

Being the model to fit:

\[
P(x | \vec{\mu}, \vec{\sigma}) = \sum_{i=1}^{K} P(w_i) \prod_{j=1}^{A} P(x | \mu_{ij}, \sigma_{ij}, w_i)
\]

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 update of the parameters (we omit the derivations) in the maximization step is:

\[
\hat{\mu}_i = \frac{\sum_{k=1}^{N} P(w_i | x_k, \vec{\mu}, \vec{\sigma}) x_k}{\sum_{k=1}^{N} P(w_i | x_k, \vec{\mu}, \vec{\sigma})}
\]

\[
\hat{\sigma}_i = \frac{\sum_{k=1}^{N} P(w_i | x_k, \vec{\mu}, \vec{\sigma})(x_k - \hat{\mu}_i)^2}{\sum_{k=1}^{N} P(w_i | x_k, \vec{\mu}, \vec{\sigma})}
\]

\[
\hat{P}(w_i) = \frac{1}{N} \sum_{k=1}^{N} P(w_i | x_k, \vec{\mu}, \vec{\sigma})
\]

The EM algorithm is initialized with a set of \( K \) initial distributions usually obtained with K-means. The \( \mu_i \) 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.

It can be seen 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 described as a Dirichlet 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 2.8 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.
2.3. PARTITIONAL CLUSTERING ALGORITHMS

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:

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

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.

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:

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

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

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

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.
2.3.2 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 [32]. 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 \( \varepsilon \)-neighborhood, as the instances that are at a distance less than \( \varepsilon \) to a given instance,

\[
N_\varepsilon(x) = \{ y \in X | d(x,y) \leq \varepsilon \}
\]

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

\[
\text{Core point} \equiv |N_\varepsilon(x)| \geq \text{MinPts}
\]

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 \( \varepsilon \) and \( \text{MinPts} \) if:

1. \( p \in N_\varepsilon(q) \)
2. \( |N_\varepsilon(q)| \geq \text{MinPts} \)

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 \( \varepsilon \) and \( \text{MinPts} \) is any subset of \( D \) that:

1. \( \forall p, q \in C \land \text{density}\_\text{reachable}(q,p) \rightarrow q \in C \)
2. \( \forall p, q \in C \text{ density}\_\text{connected}(p,q) \)

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 \( \varepsilon \) and \( \text{MinPts} \).
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 2.10 shows an example of the $\varepsilon$-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 $\varepsilon$ 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 [7] uses heuristics based on the histogram of the distances among the examples to find good values for these parameters.

The main drawback of these methods come from the cost of finding the nearest neighbors for an example. To decrease the computational cost an 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 ([42]). 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):
It defines the density function of a dataset as the sum of the influences of all examples of the dataset

\[ f_B^D(x) = \sum_{i=1}^{N} e^{-\frac{1}{2} \frac{|x-y_i|^2}{\sigma^2}} \]

This function takes the role of the \( \varepsilon \)-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:

\[ \nabla f_B^D(x) = \sum_{i=1}^{N} (x_i - x) f_B^x_i(x, x) \]

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 \( \exists k \in \mathbb{N}; d(x_k, x^*) \leq \varepsilon \). This defines a path that connects the points of a dataset to the density attractors based on the density gradient function:

\[ x^0 = x, x^i = x^{i-1} + \delta \cdot \frac{\nabla f_B^D(x^{i-1})}{\|\nabla f_B^D(x^{i-1})\|} \]

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

Two kinds of clusters can be defined, Center-defined Cluster (wrt to \( \sigma, \xi \)) 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 \( \sigma, \xi \)) 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 2.11 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 \( \sigma \) and \( \xi \) 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:

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.
2.3. PARTITIONAL CLUSTERING ALGORITHMS

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

2.3.3 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 hyperrectangular 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 [72] 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 2.12). 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 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.

Figure 2.11: DENCLUE algorithm
The algorithm of CLIQUE [5] 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 2.13). 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 2.14). Two bins are connected if they share a 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 2.15). 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.

### 2.3.4 Other 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 as the Minimum Spanning Tree (MST), Voronoi tessellation or Delaunay triangularization (see figure 2.16) 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 2.17). 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 ([46]). 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 \( (RI(C_i, C_j)) \): sum of the edges that connect the two groups divided by sum of the edges that partitions each group in two equal sized groups

- Relative Closeness \( (RC(C_i, C_j)) \): mean of the edges that connect the two groups divided by 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 2.18 presents an example of how this algorithm works.
In the latest years other methods also related to graphs but using different perspectives have gained popularity, spectral clustering and affinity clustering are two examples.

Spectral graph theory defines properties that hold the eigenvalues and eigenvectors of the adjacency matrix or Laplacian matrix of a graph. Spectral clustering [56] 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 normalizations 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: $L_{sym} = D^{-1/2}LD^{-1/2}$ 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 La-
placian matrix will have only an specific number of eigenvalues larger than zero, corresponding
to the number of clusters. Also, the eigenvectors are in a space where the clusters are more
easy to identify. This means that traditional clustering algorithms can be applied to discover
these clusters, for instance, K-means.

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:

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

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

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

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:

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

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

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 com-
putation 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 [34] is an algorithm based on message passing but related to graph par-
titioning. 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 apriori, but depends of 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:
  \[
  r(i, k) = r(i, k) - \max_{k' \neq k} \{ a(i, k') + s(i, k') \}
  \]
- The availabilities are updated as:
  \[
  a(i, k) = \min \{ 0, r(k, k) + \sum_{i' \neq \{i,k\}} \max(0, r(i', k)) \}
  \]
- The self availability \( a(k, k) \) is updated as:
  \[
  a(k, k) = \sum_{i' \neq k} \max(0, r(i', k))
  \]
- 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 2.19 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.

**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.
Figure 2.19: Affinity propagation in action. Each iteration the responsibility increases for some examples (cluster prototypes) and decreases for the rest.

Figure 2.20: Example of self-organizing map using a $3 \times 3$ grid

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 $\alpha$, 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.

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. This 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 [15], that finds the
**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 \( m_j \) (\( M \))

foreach Prototype \( m_k \in M \) do

\[ m_k = m_k + \alpha (x_i - m_k) \]

end

end

end

Path across
Path inside

Figure 2.21: Examples of SVM clustering. The labeling of the examples is determined by determining the sphere that contains each example

smallest enclosing sphere in feature space for the data using a gaussian kernel:

\[ ||\Phi(x_j) - a||^2 \leq R^2 \quad \forall j \]

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 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 2.21).

Other approaches use an algorithm similar K-means like [19] or define a more general optimization problem that looks for a set of dividing planes for the examples like Maximum Margin Clustering ([79]).
2.4 Applications

2.4.1 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 $30 \times 30 \text{Km}^2$ 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 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 $\text{minpoints}$ 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 2.22 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 2.23 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
Figure 2.23: Examples of the geographical cluster obtained using K-means, they show different localized activity profiles.
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 2.24 shows some examples of the profiles obtained using this algorithm.
Figure 2.24: Examples of the geographical cluster obtained using affinity propagation, they show other more specific behavior profiles.
Clustering Validation

After processing a dataset using a unsupervised algorithm, the first question that arises is whether an actual structure has been found on the data or 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 will not be 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 of all, 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 example, 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 the way the algorithm works, it will always result in a partition, even for random data. The same happens for hierarchical clustering, we can also suspect if cutting the dendrogram 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 known structure of the data is discovered by the chosen algorithm, but the criteria used for supervised learning can be used in this case instead.

These measures can also be used for model selection (for example the number of clusters). All 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.
3.1 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:

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

If the data is uniformly distributed, the value of $H$ will be around 0.5.

3.2 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)

3.2.1 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:

- Quadratic error/Distorsion (k-means)
  $$SSE = \sum_{k=1}^{k} \sum_{x_i \in C_k} || x_i - \mu_k ||^2$$

- Log likelihood (Mixture of gaussians/EM)
3.2. MEASURES FOR CLUSTERING VALIDATION

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.

\[
S_{W_k} = \sum_{x_i \in C_k} (x_i - \mu_k)(x_i - \mu_k)^T \\
S_{B_k} = |C_k|(\mu_k - \mu)(\mu_k - \mu)^T \\
S_{M_{k,l}} = \sum_{x_i \in C_k} \sum_{x_j \in C_l} (x_i - x_j)(x_i - x_j)^T
\]

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

\[
Tr(S_W) = \frac{1}{K} \sum_{i=1}^{K} S_{W_k} \\
Tr(S_B) = \frac{1}{K} \sum_{i=1}^{K} S_{B_k}
\]

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

\[
Det(S_M) = \frac{1}{K^2} \sum_{i=1}^{K} \sum_{j=1}^{K} S_{M_{i,j}}
\]

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)

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

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)

\[
\tilde{R} = \frac{1}{K} \sum_{i=1}^{K} R_i
\]
where

$$R_{ij} = \frac{S_{Wi} + S_{Wj}}{S_{Mi}}$$

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

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)**

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

  Where

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

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

  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.

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 ([8]) 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.

Figure 3.1 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. Figure 3.2 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.

### 3.2.2 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.

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:
3.2. MEASURES FOR CLUSTERING VALIDATION

Figure 3.1: Trace of the Within Scatter matrix, Calinski-Harabasz and Davis-Bouldin indices for a dataset with 5 well separated clusters

Figure 3.2: Trace of the Within Scatter matrix, Calinski-Harabasz and Davis-Bouldin indices for a dataset with 5 noisy non well separated clusters

- 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:

- Rand/Adjusted Rand statistic:

\[
R = \frac{(a + d)}{(a + b + c + d)}; \quad ARand = \frac{a - \frac{(a+c)(a+b)}{a+b+c+d}}{\frac{(a+c)+(a+b)}{2} - \frac{(a+b)(a+c)}{a+b+c+d}}
\]

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:

\[
J = \frac{a}{(a + b + c)}
\]
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.

- Folkes and Mallow index:

\[ FM = \sqrt{\frac{a}{a+b} \cdot \frac{a}{a+c}} \]

This index uses the geometric mean instead of the arithmetic.

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:

\[ MI(Y_i, Y_k) = \sum_{X^i_c \in Y_i} \sum_{X^k_c' \in Y_k} \frac{|X^i_c \cap X^k_c'|}{N} \log_2 \left( \frac{N |X^i_c \cap X^k_c'|}{|X^i_c| |X^k_c'|} \right) \]

and the Entropy of a partition as:

\[ H(Y_i) = - \sum_{X^i_c \in Y_i} \frac{|X^i_c|}{N} \log_2 \left( \frac{|X^i_c|}{N} \right) \]

where \( X^i_c \cap X^k_c' \) 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:

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

### 3.2.3 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:
3.3 Assessing the number of clusters

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 3.2.1 can be used for this purpose as shown in figures 3.1 and 3.2, for instance the Calinski 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:**
  
  $$H(k) = \left[ \frac{S_W(k)}{S_W(k + 1)} - 1 \right] (n - k - 1)$$
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:
  \[
  KL(k) = \frac{\text{DIFF}(k)}{\text{DIFF}(k+1)}
  \]
  with \( \text{DIFF}(k) = (k-1)^{2/p} S_W(k-1) - k^{2/p} S_W(k) \)

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.

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.

A very successful test that uses this method is the Gap statistic ([69]), defined as
\[
\text{Gap}(k) = \frac{1}{B} \sum_b \log(S_W(k)_b) - \log(S_W(k))
\]

From the standard deviation \( sd_k \) of \( \sum_b \log(S_W(k)_b) \) is defined \( s_k \) as:
\[
s_k = sd_k \sqrt{1 + 1/B}
\]

The probable number of clusters is the smallest number that holds:
\[
\text{Gap}(k) \geq \text{Gap}(k+1) - s_{k+1}
\]

A third method is to use the concept of cluster stability ([51, 59]). 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.
3.4 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 inorder 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 3.3 shows the distance matrix of three datasets. The first one corresponds to five well separated clusters, it can be seen 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 3.4 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

Figure 3.3: Distance matrix visualization. The first plot correspond to a dataset with 5 well separated clusters, the second one to 5 overlapping clusters and the third one to random generated data
how to proceed when processing a dataset.

3.5 Applications

3.5.1 Wheelchair control

This example continues the application described in section 1.5.2. 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 3.5) 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 3.6 (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 3.6 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 3.5: Visualization of the three principal components obtained from PCA

Figure 3.6: Plot of the silhouette, adjusted rand and normalized mutual information indices for different number of clusters for the wheelchair dataset
Clustering Large Datasets

According to a recent poll about the most frequent tasks and methods employed in data mining projects (KDNuggets, 2011), 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 has shifted from tens of gigabytes in 2011 to terabytes in 2013. 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 on-line 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.

4.1 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 can be used in combination. These strategies can be classified in:

**One-pass strategies:** The constraint assumed is that the data only can be processed once and in a sequential fashion. 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 to obtain 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.

### 4.1.1 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 to store 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 these 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.

### 4.1.2 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.

### 4.1.3 Sampling/batch strategies

The purpose of sampling and batch strategies is to allow to perform 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 assume 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.

### 4.1.4 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.

### 4.1.5 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 rely sometimes 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 merge 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.
4.2 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.

4.2.1 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 [60] 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.

4.2.2 Rough-DBSCAN

In [71] 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. This 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 $\tau$ 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:

$$k \leq \frac{V_S}{V_{\tau/2}}$$

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 $\tau$, 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 $\epsilon$ 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 $\epsilon$ to the leader $l$:

$$\mathcal{L}_l = \{l_j \in \mathcal{L} \mid \|l_j - l\| \leq \epsilon\}$$

The measure $\text{roughcard}(N_\epsilon(l, \mathcal{D}))$ is defined as:

$$\text{roughcard}(N_\epsilon(l, \mathcal{D})) = \sum_{l_i \in \mathcal{L}_l} \text{count}(l_i)$$

approximating the number of examples less than a distance $\epsilon$ 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.

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 $\tau$ 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.

### 4.2.3 CURE

CURE [38] 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 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 $O\left(\frac{n^2}{p} \log\left(\frac{n}{p}\right)\right)$, being $n$ the size of the sample and $p$ the number of batches used.

### 4.2.4 BIRCH

BIRCH [80] 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 4.2) is a balanced n-ary tree that contains information that represents probabilistic prototypes. Leaves of the tree can contain as much as \( l \) prototypes and their radius cannot 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 whether the distance is greater or not than the value of the parameter \( t \). If the current leave has no 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
4.2. ALGORITHMS

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 dataset has a cost $O(nk)$, being $k$ the number of clusters.

4.2.5 OptiGrid

OptiGrid [43] 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 4.1

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 $O(Nk)$, $O(Ndk)$
Algorithm 4.1 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 $\{P_1(X), ..., P_k(X)\}$

$\text{BestCuts} \leftarrow \emptyset$, $\text{Cut} \leftarrow \emptyset$

for $i \in 1..k$ do
  $\text{Cut} \leftarrow \text{ComputeCuts}(P_i(X))$
  for $c$ in $\text{Cut}$ do
    if $\text{CutScore}(c) > \min_{c,q}$ then $\text{BestCuts}.append(c)$
  end
end

if $\text{BestCuts}.isEmpty()$ then return $X$ as a cluster

$\text{BestCuts} \leftarrow \text{KeepQBestCuts}(\text{BestCuts},q)$

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

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 $O(Nd \log(N))$

### 4.2.6 Scalable K-means

This early algorithm for clustering scalability presented in [17] combines a sampling strategy and a summarization strategy. The main purpose of this algorithm is to provide an on-line 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 to reduce the amount of data needed to maintain the model each iteration.
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 form 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.

4.2.7 STREAM LSEARCH

The STREAM LSEARCH algorithm [37] 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 not 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.

### 4.2.8 Mini batch K-means

Mini Batch K-means [66] 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
4.2. ALGORITHMS

**Algorithm 4.2** Mini Batch K-Means algorithm

Given: k, mini-batch size b, iterations t, data set X

Initialize each \( c \in C \) with an \( x \) picked randomly from \( X \)

\[ v \leftarrow 0 \]

for \( i \leftarrow 1 \) to \( t \) do

\[ M \leftarrow b \text{ examples picked randomly from } X \]

for \( x \in M \) do

\[ d[x] \leftarrow f(C,x) \]

end

for \( x \in M \) do

\[ c \leftarrow d[x] \]

\[ v[c] \leftarrow v[c] + 1 \]

\[ \eta \leftarrow \frac{1}{v[c]} \]

\[ c \leftarrow (1-\eta)c+\eta x \]

end

end

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 4.2.

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 \(O(kni)\).

### 4.2.9 Canopy clustering

Canopy clustering [58] 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 \((T_1)\) 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 4.5 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 4.2.2, 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 computational cost of applying this algorithm to $k$ canopies will be $O(\frac{n^2}{k})$.

4.2.10 Indexed K-means

The proposal in [45] relies in 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,
4.2. ALGORITHMS

KD-TREE

Figure 4.6: Indexed K-means

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 leaf 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 4.6.

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 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.
4.2.11 Quantized K-means

The proposal in [77] 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 multi-dimensional 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 $\lambda_l = \frac{m - p_l}{\rho}$, being $p_l$ 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 this 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 4.7, 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 $d(b_i, s_*)$. Then, for each prototype the minimum distance to all the bins is computed and the prototypes that are at a distance less than $d(b_i, s_*)$ 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, a large number of bins will be empty, reducing the actual number of computations.
Sometimes we have available additional information about the dataset that we are analyzing, other than only the examples. It could be interesting to incorporate this information to the clustering process in order to bias the search that performs the algorithm we are using towards a solution more consistent with our knowledge and to improve the quality of the result by reducing the algorithm natural bias (predictivity/stability). It can be though like a regularization term added to the data that depends on domain information, or a set of priors about the model, if we are using a probabilistic approach.

The information that we could have available can be of different kinds, including sets of labeled instances, constrains among a subset of examples, indicating instances that have to be in the same group or instances that can not belong to the same group, or, also, general information about the properties that the instances of a group have to hold.

The use we make of this information will depend on the type of model used by the algorithm we are applying, we could:

- begin with a prior model that changes how the search is performed
- bias the search, pruning the models that are not consistent with the available knowledge
- modify the similarity among instances to match the constraints imposed by the prior knowledge

The methods used by semisupervised clustering are different from the ones used by semisupervised learning. In that area there is always a set of labeled examples and unlabeled data is used to improve the performance of supervised learning. For semisupervised clustering the classes in the data are not fixed, they have to be discovered and the use of labeled data only describes a partial view of the classes.

### 5.1 Biasing using labeled examples

Assuming that we have some labeled examples, these can be used to obtain an initial model. We only have to know what examples belong to some clusters, the actual ones are not needed. We obtain an initial model to initialize the clustering process and there is not guarantee that, in the end, the labeled data will belong to the same clusters.

This means that we only decide the starting point of the search using the supervised information. This initial model changes the search of the learning algorithm and the final model (bias).
An example of this methodology is the variation of the K-means algorithm described in [13]. The usual initialization of K-means is by selecting randomly the initial prototype. The semisupervised approach changes the way the initial centers are chosen. The paper proposes two methods:

- To use the labeled examples to build the initial prototypes (seeding), but allowing that successive iterations move the examples to their natural prototypes

- To use the labeled examples to build the initial prototypes and constrain the model so the labeled examples are always in the initial clusters (seed and constraint)

In algorithm 5.1 both methods are detailed. The initial prototypes work as prior for the model/probability distribution of the clusters. The same procedure can be applied to other algorithms that start with an initial estimation of the clusters like for instance GMM.

5.2 Biasing using instance constraints

To have labeled examples means that the number of clusters and something about the characteristic of the data is known. Sometimes it is easier to have information about if two examples have to be in the same or different clusters. Imagine for instance that we are segmenting an image with different objects in it, it would be easy to give information about some pixels that belong to the same object and information about some pixels that are from different objects.
This information can be expressed by means of constraints among examples: *must links* and *cannot links*. Figure 5.1 represents an example of these constraints. This information can be used as a regularization term to bias the search and only look for models that maintain these constraints.

A method following this approach is described in \[14\]. It is also an algorithm based on K-means (spherical clusters based on prototypes). A set of must-link cannot-link constraints is defined over a subset of examples. The quality function of the K-means algorithm is modified to bias the search including as a penalization how well the model maintains the constraints. For this a hidden markov random field is defined using the constraints.

The labels of the examples are used to define the variables in the markov random field. The must-links and cannot-links define the dependence among the variables. The clustering of the examples has to maximize the probability of the hidden markov random field (see figure 5.2).

A new objective function for the K-Means algorithm is defined. The main idea is to introduce a penalty term to the objective function that:

- Penalizes the clustering that assigns examples with must-links in different clusters
Algorithm 5.2 HMRF-KMeans Algorithm

**Algorithm:** HMRF-KMeans

**Input:** The dataset $\mathcal{X}$, the number of clusters $K$, a set of must and cannot links, a distance function $D$ and a set of weights for violating the constraints

**Output:** A partition of $\mathcal{X}$ in $K$ groups

begin
  Compute $K$ initial prototypes ($\mu_i$) using constraints
  repeat
    **E-step:** Reassign the labels of the examples using the prototypes ($\mu_i$) to minimize $J_{obj}$
    **M-step:** Given the cluster labels recalculate cluster centroids to minimize $J_{obj}$
  until Convergence
end

- Penalizes the clustering that assigns examples with cannot-links in the same cluster

This penalty has to be proportional to the distance among the examples. The new objective function is:

$$J_{obj} = \sum_{x_i \in X} D(x_i, \mu_i) + \sum_{(x_i, x_j) \in \mathcal{M}, [l_i \neq l_j]} w_{ij} \varphi_D(x_i, x_j) + \sum_{(x_i, x_j) \in \mathcal{C}, [l_i = l_j]} \bar{w}_{ij} (\varphi_{D_{max}} - \varphi_D(x_i, x_j))$$

Where $D$ is the distortion function used by the clustering algorithm, $\mathcal{M}$ and $\mathcal{C}$ are the sets of must and cannot links, $w_{ij}$ and $\bar{w}_{ij}$ are the weights of the penalty for violating the constraints and $\varphi_D$ is an increasing function of the distance between two examples. The specific function depends on the distortion measure selected.

The initialization of the process begins by analyzing the constraints and inferring a set of examples to obtain the initial prototypes. The constraints conform a graph that divides a subset of the examples into different groups (connected components of the graph). With these instances $K$ prototypes are computed and used to initialize the algorithm. The algorithm minimizes the objective function using an expectation and a minimization step. Algorithm 5.2 gives a description of the procedure.

### 5.3 Distance Learning

The constraints can also be seen as an indication of the inadequacy of the distance measure between two examples. A must-link violation means that the distance function assigns a similarity larger than the desired similarity. A cannot-link violation means that distance function assigns a similarity smaller than the desired similarity. For instance, the algorithm presented in the previous section can be modified to introduce additional weights that change the distances among examples involved in the constraints (an additional maximization step is needed).

Other approaches directly learn the most adequate distance function to fulfill the constraints. They are used as a guide to find a distance matrix that represents better the relations among examples. This can be defined as an optimization problem that optimizes the distances among examples with respect to the constraints. This approach is related to kernel methods, the goal is to learn a kernel matrix that represents a new space where the instances have appropriate distances.
5.3. DISTANCE LEARNING

Figure 5.3: Example of semisupervised clustering using a distance learning method

Figure 5.3 show an example of distance learning, initially, the constraints indicate pairs of examples that must or cannot be in the same class. Each iteration the distance among the examples is changed learning a transformation matrix (similarly to dimensionality reduction) that separate/approach examples according to their constraints. This distance affects not only to the examples included in the constraints, but also to the rest of the examples, so a new space is learned for all the data.

This is a current research area that has developed different approaches, for instance:

- **Relevant Component Analysis** ([76]), optimization of linear combination of distance kernels generated by must and cannot links

- Optimization using Spectral Graphs for maintaining the constrains in the new space and the structure of the original space ([54])

- Learning of Mahalanobis distances: separate/approach the different dimensions to match the constraints ([11])

- Kernel Clustering (Kernel K-means) with kernel matrix learning via regularization ([10])

- Probabilistic Clustering (Model Based) with constraints via EM, modifying the probabilities matching the constrains ([55])

An excellent survey on distance learning can be found in [50].
Association Rules

There are applications where we are interested in the relations among the different variables that describe the data. The idea is that these relations can reveal some notion of causality through attribute correlation.

There are different models focused on discovering these kinds of relationships like bayesian networks for instance. Even clustering methods can be applied for finding these relationships. In this chapter we will focus on a more simple model, but that is in essence related to the techniques from these methods.

The idea is to lose focus on the examples in the data and to use them only for assessing variable relationships. The methods that we are going to describe focus specifically on discrete valued variables, and were initially described for binary data, but they can be easily applied to multivalued discrete data using binarization as preprocess.

A binary dataset is a collection of examples where each row is described by a binary attribute. For instance:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>T3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>T4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>

The usual meaning for the variable is that a specific event occurs or not for an example. The typical domain does not allow having many attributes for an example, for instance, individual purchase transactions from a supermarket. This makes easy to represent the data as a sparse matrix that only stores the occurrences of the attributes. For instance, the database defines individual purchases of items by clients:

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bread, Chips, Beer, Yogourt, Eggs</td>
</tr>
<tr>
<td>2</td>
<td>Flour, Beer, Eggs, Butter,</td>
</tr>
<tr>
<td>3</td>
<td>Bread, Ham, Eggs, Butter, Milk</td>
</tr>
<tr>
<td>4</td>
<td>Flour, Eggs, Butter, Chocolate</td>
</tr>
<tr>
<td>5</td>
<td>Beer, Chips, Bacon, Eggs</td>
</tr>
</tbody>
</table>

For now we will consider that each example is independent from the rest, only considering the link among variables, looking patterns called association rules. In the next chapter we will add a temporal link to the data considering sequences of transactions from the same user.
6.1 Definitions

From a binary/transactions database we will define an association rule as the representation of the coocurrence of events in the database. For instance, in the previous example of database we can find that the following associations appear several times:

\[
\{\text{Flour}\} \rightarrow \{\text{Eggs}\} \\
\{\text{Beer}\} \rightarrow \{\text{Chips}\} \\
\{\text{Bacon}\} \rightarrow \{\text{Eggs}\}
\]

We are going to be interested on the associations that appear with a specific frequency. Formally, we will define \( R \) as the set of attributes that a transaction of the database can have. We define also \( X \) as a subset of attributes from \( R \). We say that \( X \) is a pattern in the database if there is any row where all the attributes of \( X \) are present (its value is 1).

We define the support (frequency) of a pattern \( X \) as the function:

\[
fr(X) = \frac{|M(X,r)|}{|r|}
\]

where \( |M(X,r)| \) is the number of times that \( X \) appears in the DB and \( |r| \) is the size of the database. Given a minimum support \((\text{min}\_\text{sup} \in [0,1])\), we say that the pattern \( X \) is frequent (frequent itemset) if:

\[
fr(X) \geq \text{min}\_\text{sup}
\]

\( \mathcal{F}(r, \text{min}\_\text{sup}) \) is the set of patterns that are frequent in the database:

\[
\mathcal{F}(r, \text{min}\_\text{sup}) = \{X \subseteq R / fr(X) \geq \text{min}\_\text{sup}\}
\]

Given a database with \( R \) attributes and \( X \) and \( Y \) attribute subsets, with \( X \cap Y = \emptyset \), an association rule is the expression:

\[
X \Rightarrow Y
\]

The function \( conf(X \Rightarrow Y) \) is the confidence of an association rule, computed as:

\[
conf(X \Rightarrow Y) = \frac{fr(X \cup Y)}{fr(X)}
\]

We consider a minimum value of confidence \((\text{min}\_\text{conf} \in [0,1])\). Given a minimum confidence and a minimum support, an association rule \((X \Rightarrow Y)\) exists in a DB if:

\[
(fr(X \cup Y) \geq \text{min}\_\text{sup}) \land (conf(X \Rightarrow Y) \geq \text{min}\_\text{conf})
\]

There is a trivial approach for obtaining all the association rules that have a support and confidence higher than given thresholds, it is enough to find all frequent subsets from \( R \). This means to explore for all possible patterns \( X \) with \( X \subseteq R \) and subpatterns \( Y \) where \( Y \subseteq X \) the association rule \((X - Y) \Rightarrow Y\). Obviously, there are \( 2^{|R|} \) possible candidates, making this approach unpractical. Figure 6.1 represent all possible subsets considering four attributes.
6.2. The Apriori Algorithm

We need an approach that prunes the search space, reducing the set of candidates to a more practical size. Given \( X \) and \( Y \), with \( Y \subseteq X \), the following property holds:

\[
\text{if } fr(Y) \geq fr(X), \text{ if } X \text{ is frequent, } Y \text{ also is frequent.}
\]

In other words, if any subset \( Y \) from \( X \) it is not frequent then \( X \) is not frequent. This is known as the anti-monotone property of support.

A feasible exploration approach is to begin with all the frequent sets of size 1 and to increase its size iteratively, checking the anti-monotone property in order to prune the candidates that include infrequent itemsets. Figure 6.2 represents the effect of pruning all subsets that contain the itemset CD considering that is not frequent.

The \textit{apriori} algorithm was the first one developed for the exploration of the space of association rules. It defines the set of candidates to evaluate each iteration in the following way, being \( \mathcal{F}_l(r, \text{min}\_\text{sup}) \) the set of frequent sets from \( R \) of size \( l \), and given a set of patterns of length \( l \), the only frequent set candidates of length \( l+1 \) will be those which all subsets are in the frequent sets of length \( l \). Formally:

\[
C(\mathcal{F}_{l+1}(r)) = \{ X \subseteq R/|X| = l + 1 \land \forall Y \ (Y \subseteq X \land |Y| = l \Rightarrow Y \in \mathcal{F}_l(r))\}
\]
Algorithm 6.1 Apriori algorithm for frequent itemsets exploration

**Algorithm:** Apriori \((R, \text{min}_s\text{up}, \text{min}_c\text{onf})\)

\(C, CC\text{an}, C\text{Temp}:\) set of frequent subsets

\(R\text{As}:\) set of association rules, \(L: \) integer

\(L=1\)

\(CC\text{an}=\) Frequent sets \(1(R, fr\text{_min})\)

\(R\text{As}=\emptyset\)

while \(CC\text{an} \neq \emptyset\) do

\(L=L+1\)

\(C\text{Temp}=\) Candidate sets \(L(R, CC\text{an}) = C(\mathcal{F}_{l+1}(r))\)

\(C=\) Frequent sets \(C\text{Temp}, \text{min}_s\text{up}\)

\(R\text{As}=R\text{As} \cup \) Confidence rules \((C, \text{min}_c\text{onf})\)

\(CC\text{an}=C\)

end

This means that the computation of association rules can be done iteratively starting with the smallest frequent subsets until no more candidates are obtained. Algorithm 6.1 presents an implementation.

For example, given four attributes \(\{A, B, C, D\}\), and a database of transactions we have a set of possible candidates (see figure 6.1). If we compute the pattern candidates of length 1, we will find that some attributes will have a frequency larger than a specific support threshold. Assume that these attributes are \(\{A, C, D\}\), the next graph shows the space of candidates after pruning candidates that include the non-frequent item \(B\).

The patterns of length 2 that are not pruned will be the possible candidates for frequent itemsets of this length. If we assume that all these patterns are frequent we will have that there is only one pattern left as candidate for frequent patterns of length three, so the algorithms stops at this pattern length.

Something that has to be noticed is that the specific value used as \(\text{min}_s\text{up}\) has effect on the computational cost of the search, because it limits what is going to be explored. If it is too high, only a few patterns will appear (we could miss interesting rare occurrences), if it is too low the computational cost will be too high (too many associations will appear). The extreme is when the value is 0, in this case all items are frequent, so all the possible combinations of items will be generated. Unfortunately the adequate threshold value for a given application
6.2. THE APRIORI ALGORITHM

Sometimes, multiple minimum supports could be used, different for different groups of items, to explore the interesting associations. This could reduce the combinatorial explosion due to very frequent items and help focus on less frequent but more interesting ones. These thresholds are domain specific and some previous analysis of the data should be performed to guess these values.

Other elements of the problem also have an influence on the computational cost, such as the number of different items, the more items there are, the more space is needed to count its support, or the size of the database, because the algorithm has to perform multiple passes to count the support, or the average transaction width, that affects the maximum length of frequent itemsets.

Some specialized algorithms have been developed to deal with these problems in some domains, for example, in bioinformatics, it is usual to have long frequent itemsets.

There are variations of the way the itemsets are explored by the apriori algorithm that can reduce the computational cost depending on the characteristics of the items. The strategy followed by the algorithm explores the lattice of itemsets from general-to-specific, but the items can be explored from specific-to-general when the maximum itemset size is large or the transactions are dense. In this case an item has to be decomposed in all its subsets if it is not frequent, and exploration stops for a candidate when it is maximal frequent. A didirectional search can also be applied using the information from one direction to prune candidates from the other direction. Figure 6.3 represents all three strategies.

Other methods for reducing the computational cost include defining equivalence classes for the items, so several items are counted as the same one, or using prefixes (or suffixes) as equivalence classes when it makes sense in the domain.

Also, the way the itemsets are explored can be changed from a breadth first search (as apriori has been defined) that can be memory demanding, to a depth first search, that requires less memory for the exploration. This strategy is exploited by algorithms targeting maximal frequent itemsets or that use a specific-to-general exploration of the itemsets. The purpose is to detect earlier maximal itemsets, so more pruning can be applied to the search.

Despite of pruning and search strategies for Association Rules candidate generation is expensive. In the next section we will introduce other strategies that allow extracting association rules using specialized data structures that avoid the exhaustive generation of candidates.

An additional problem of frequent itemsets mining is that the number of possible association rules obtained by the apriori algorithm can be very large. This can be improved by targeting subsets of itemsets that have specific properties. There are two kinds of itemsets that are used

Figure 6.3: Different search strategies for the apriori algorithm
for reducing the number of candidates:

- **Maximal frequent itemsets**, defined as frequent itemset for which none of its immediate supersets are frequent

- **Closed frequent itemsets**: defined as frequent itemset for which none of its immediate supersets have the same support

This is the relationship among these subsets and the whole set of frequent itemsets.

\[ \text{Maximal frequent itemsets} \subseteq \text{Closed frequent itemsets} \subseteq \text{Frequent itemsets} \]

The actual reduction on the number of final item sets will depend on the domain and the support threshold, but in practice the number of candidates in these subsets are sometimes orders of magnitude smaller.

### 6.2.1 Example

To illustrate how the apriori algorithm works we will extract the frequent itemsets from the following dataset of transactions:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

First we have to give the minimum support threshold, for example we can decide arbitrarily to use a value of 0.25. In this case there is no information from the domain that helps us to decide the value, but obviously a deeper study of the data is necessary to make this decision. In this case we can observe that all columns have larger frequency, thus, the set of candidates of length 1 is:

\[ \text{F1} = \{[A],[B],[C],[D],[E],[F]\} \]

From this we can compute the candidates for frequent sets of length 2 (all combinations):
6.2. THE APRIORI ALGORITHM

\[ C(F1) = \{[A, B], [A, C], [A, D], [A, E], [A, F], [B, C], [B, D], [B, E], [B, F], [C, D], [C, E], [C, F], [D, E], [D, F], [E, F]\} \]

From the data table we can compute the frequency of each candidate:

\[
\begin{align*}
fr([A, E]) &= 10/20 & fr([B, F]) &= 4/20 \\
fr([A, F]) &= 4/20
\end{align*}
\]

Of the candidates, the ones that have a support greater than 25% are:

\[ F_2 = \{[A, B], [A, C], [A, E], [B, C], [B, E], [C, F]\} \]

In the lattice exploration we would have:

\[
\begin{array}{c}
\text{AB} & \text{AC} & \text{AD} & \text{AE} & \text{AF} & \text{BC} & \text{BE} & \text{BD} & \text{BF} & \text{CD} & \text{CE} & \text{CF} & \text{DE} & \text{DF} & \text{EF} \\
\text{FR}=25\%
\end{array}
\]

If we evaluate the confidence of each one of the association rules we have

\[
\begin{align*}
\text{conf}(A \rightarrow B) &= 11/15 & \text{conf}(A \rightarrow E) &= 10/15 & \text{conf}(B \rightarrow E) &= 11/16 \\
\text{conf}(B \rightarrow A) &= 11/16 & \text{conf}(E \rightarrow A) &= 10/13 & \text{conf}(E \rightarrow B) &= 11/13 \\
\text{conf}(A \rightarrow C) &= 6/15 & \text{conf}(B \rightarrow C) &= 6/16 & \text{conf}(C \rightarrow F) &= 5/9 \\
\text{conf}(C \rightarrow A) &= 6/9 & \text{conf}(C \rightarrow B) &= 6/9 & \text{conf}(F \rightarrow C) &= 5/13
\end{align*}
\]

Choosing also arbitrarily a confidence threshold of 2/3, the rules that are significant are:

\[ \{A \rightarrow B, B \rightarrow A, C \rightarrow A, A \rightarrow E, E \rightarrow A, C \rightarrow B, B \rightarrow E, E \rightarrow B\} \]

From the set \( F_2 \), we can compute the set of candidates for \( F_3 \)

\[ C(F2) = \{[A, B, C], [A, B, E]\} \]

\[
\begin{align*}
fr([A, B, C]) &= 3/20 & fr([A, B, E]) &= 8/20
\end{align*}
\]

Of the candidates, the ones that have support greater than 25% are:
The confidence of their corresponding association rules are:

\[
\text{conf}(A \land B \rightarrow E) = \frac{8}{11} \\
\text{conf}(A \land E \rightarrow B) = \frac{8}{10} \\
\text{conf}(B \land E \rightarrow A) = \frac{8}{11}
\]

If we choose as confidence value 2/3 the rules that are significant:

\{A \land B \rightarrow E, A \land E \rightarrow B, B \land E \rightarrow A\}

From all the itemsets discovered:

\begin{align*}
\text{fr}([A,B]) &= \frac{11}{20} \\
\text{fr}([B,C]) &= \frac{6}{20} \\
\text{fr}([A,C]) &= \frac{6}{20} \\
\text{fr}([B,E]) &= \frac{11}{20} \\
\text{fr}([A,E]) &= \frac{10}{20} \\
\text{fr}([A,B,E]) &= \frac{8}{20}
\end{align*}

\{[A,C], [B,C], [C,F], [A,B,E]\} are maximal frequent
\{[A,B],[B,E],[A,E]\} are closed frequent

Now we choose a minimum support of 50%. The columns that have larger frequency are only A, B and E. Therefore:

\[F_1 = \{[A],[B],[E]\}\]

From this we can compute the support set candidates:

\[C(F_1) = \{[A,B],[A,E],[B,E]\}\]

All have a support greater than 50%, therefore

\[F_2 = \{[A,B],[A,E],[B,E]\}\]

And the candidates are:

\[C(F_2) = \{[A,B,E]\}\]

That also have a support greater than 50%, thus:

\[F_3 = \{[A,B,E]\}\]
6.3 Mining without candidate generation

The problem of the apriori approach is that the number of candidates to explore in order to find long patterns can be very large. The approach of FP-Growth [39] tries to obtain patterns from transaction databases without candidate generation. Its algorithm is based on a specialized data structure called FP-Tree that summarizes the frequency of the patterns in the patterns database. This allows exploring the patterns incrementally by adding prefixes and without candidate generation.

One of the goals of this structure is to avoid the expensive process of querying the database to compute the frequency of patterns. The representation used to explore the transactions only needs to store the items that are present. It is also assumed that there is a total ordering among the items. The resulting patterns are not dependent on this order. This order allows obtaining common prefixes from the transactions that can be used to store efficiently their frequencies. The transactions with common prefixes are merged inside the structure, maintaining this way the frequency of each subprefix. This allows later computing the frequent patterns.

The algorithm to insert the transactions of a database in the FP-tree is the following:

1. Compute the frequency of each individual item in the DB
2. Create the tree root (empty prefix)
3. For each transaction from the DB
   (a) Pick the more frequent items
   (b) Order the items by their original frequency
   (c) Iterate for each item, inserting it in the tree
       • If the node has a descendant equal to the actual item increase its frequency
       • Otherwise, a new node is created

For instance, lets assume that we have the following database of transactions with items \{a, b, c, d, e, f\}:

<table>
<thead>
<tr>
<th>N Trans</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, e, f</td>
</tr>
<tr>
<td>2</td>
<td>a, c, b, f, g</td>
</tr>
<tr>
<td>3</td>
<td>b, a, e, d</td>
</tr>
<tr>
<td>4</td>
<td>d, e, a, b, c</td>
</tr>
<tr>
<td>5</td>
<td>c, d, f, b</td>
</tr>
<tr>
<td>6</td>
<td>c, f, a, d</td>
</tr>
</tbody>
</table>

We first compute the number of times of each one of the distinct items appears in the database, we have the following counts:

<table>
<thead>
<tr>
<th>Item</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td>3</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
</tr>
</tbody>
</table>
Now we can select only the items that have a count larger than a specific threshold, in this case we select 4, and then recompute the transactions leaving only the items that have a count above that threshold:

<table>
<thead>
<tr>
<th>N Trans</th>
<th>Items (fr=4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, f</td>
</tr>
<tr>
<td>2</td>
<td>a, c, b, f</td>
</tr>
<tr>
<td>3</td>
<td>a, b, d</td>
</tr>
<tr>
<td>4</td>
<td>a, b, c, d</td>
</tr>
<tr>
<td>5</td>
<td>b, c, d</td>
</tr>
<tr>
<td>6</td>
<td>a, c, d, f</td>
</tr>
</tbody>
</table>

Now we can apply the algorithm to insert the transactions in the FP-Tree structure. The structure links the items of the transactions with the stored prefixes, so later it can be queried about the counts of transactions that contain a specific item. The first transaction to insert is \{a, f\}, because there is not other transaction in the structure it creates a branch of the tree with a count of 1 for each item:

```
Trans(a,f)

5 a
4 b
4 c
4 d
4 f
```

The second transaction is \{a, c, b, f\}, the item a is already in the root of the tree, so this increments its counting, the next item is c that is not in this level of the tree, so a new branch is created with a count of 1 for each remaining item:

```
Trans(a,b,c,f)

5 a
4 b
4 c
4 d
4 f
```

The third transaction is \{a, b, d\}, now we have the common prefix ab, so the counts of these items are increased and b adds d a new descendant with count 1:
The fourth transaction is \{a, b, c, d\}, now we have the common prefix abc, so the counts of these items are increased and c adds d a new descendant with count 1:

The fifth transaction is \{b, c, d\}, now we do not have any common prefix in the structure, so a new branch is created in the tree for b at the top level and all the items of the transaction are inserted under this new branch with count 1:

Finally, the last transaction is \{a, c, d, f\}, that has only a as common prefix in the tree, this increases its count, and a new branch is inserted in a for c, adding the rest of the items of the transaction with count 1:
This final tree shows the actual links between the items and their positions in the different branches of the tree.

From the FP-tree we can obtain all frequent patterns using its properties and a bottom-up strategy:

- For any given frequent item \( a_i \) we can obtain all the frequent patterns that contain the item following the links of \( a_i \) in the tree
- To compute the frequent patterns with \( a_i \) as suffix only the prefix subpaths of the nodes \( a_i \) have to be used and their frequency corresponds to the frequency of node \( a_i \)

The frequent patterns can be obtained from a recursive traversal of the paths in the FP-tree and the combination of subpatterns.

1. Given an item, select all the paths that contain the item (prefix paths)
2. Convert all the paths that contain the item into a conditional FP-tree:
   (a) Update the counts along the path using the frequency of the selected item
   (b) Truncate the paths removing the nodes for the item
   (c) Eliminate from the paths the items that are no longer frequent (if any)
3. For all the items previous in order that are frequent in the conditional FP-tree
   (a) Count the prefix as frequent
   (b) Recursively find the frequent items for that prefix

A conditional FP-tree is a FP-Tree projected on the item that we are generating the query. This is a FP-Tree that only contains the branches of the original FP-Tree that have the item with a frequency larger that the minimum support.

For instance, we want to obtain all the patterns with suffix d from the following FP-tree, assuming a minimum support of 2:
First we keep only the paths that contain \(d\) (\(\text{minsupport} = 2\)), eliminating all the branches that does not contain the item that we are querying:

\[
\begin{array}{c}
\text{a:10} \\
\text{b:7} \\
\text{c:4} \\
\text{b:5} \\
\text{c:4} \\
\text{d:2} \\
\text{d:2} \\
\text{d:2} \\
\text{c:4} \\
\text{d:3}
\end{array}
\]

Now we generate the **conditional FP-tree** for \(d\). We update the counts of the branches according to the counts of \(d\), for instance the top level item \(b\) includes two branches with the item \(d\), both with 2 occurrences, so the new count of \(b\) is 4:

\[
\begin{array}{c}
\text{a:2} \\
\text{b:4} \\
\text{c:3} \\
\text{b:2} \\
\text{c:2} \\
\text{d:2} \\
\text{d:2} \\
\text{d:2} \\
\text{c:3} \\
\text{d:3}
\end{array}
\]

Now we can prune the paths eliminating \(d\), that is no longer needed because all patterns will have this item:

\[
\begin{array}{c}
\text{a:2} \\
\text{b:4} \\
\text{c:3} \\
\text{b:2} \\
\text{c:2}
\end{array}
\]

Finally, we solve the problem for the predecesors of \(d\), in this case \(b\) and \(c\), we are looking if \(bd\) and \(cd\) are frequent:

\[
\begin{array}{c}
\text{a:2} \\
\text{b:4} \\
\text{c:3} \\
\text{b:2} \\
\text{c:2}
\end{array}
\]

Notice that at this point these two problems can be solved in parallel, given that the parts of the FP-Tree that contain the counts are disjoint. If we continue the algorithm, the patterns extracted would be \{[d], [bd], [cd], [abd], [bcd]\}.
6.4 Measure for association rules interestingness

One of the crucial points of association rules mining is to determine what rules are relevant. This is measured using functions that score the interestingness of a rule. Given a rule, its interestingness can be computed from a contingency table of their elements:

<table>
<thead>
<tr>
<th></th>
<th>Y</th>
<th>¬Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>(f_{11})</td>
<td>(f_{10})</td>
</tr>
<tr>
<td>¬X</td>
<td>(f_{01})</td>
<td>(f_{00})</td>
</tr>
<tr>
<td></td>
<td>(f_{+1})</td>
<td>(f_{+0})</td>
</tr>
</tbody>
</table>

- \(f_{11}\); support of \(X\) and \(Y\)
- \(f_{10}\); support of \(X\) and \(¬Y\)
- \(f_{01}\); support of \(¬X\) and \(Y\)
- \(f_{01}\); support of \(¬X\) and \(¬Y\)

Until now we have only used the confidence of a rule \(\frac{f_{11}}{f_{1+}}\) to decide if a rule is selected, but sometimes this measure does not represent well its interestingness, for instance, given the following counts:

<table>
<thead>
<tr>
<th></th>
<th>Y</th>
<th>¬Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>¬X</td>
<td>75</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>10</td>
</tr>
</tbody>
</table>

\(\text{conf}(X \rightarrow Y) = 0.75\) but \(\text{conf}(¬X \rightarrow Y) = 0.9357\), so it seems that actually there are more confidence in that the items are negatively correlated.

There are other measures in the literature based on probabilistic independence and correlation that try to give a better approximation of the interestingness of a rule using all the information provided by the counts of that contingency table. The following table is a small selection of these measures:

<table>
<thead>
<tr>
<th>Lift/Interest factor</th>
<th>(N \times \frac{f_{11}}{f_{1+} \times f_{1+}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Confidence</td>
<td>(\min\left(\frac{f_{11}}{f_{1+}}, \frac{f_{11}}{f_{1+}}\right))</td>
</tr>
<tr>
<td>Max Confidence</td>
<td>(\max\left(\frac{f_{11}}{f_{1+}}, \frac{f_{11}}{f_{1+}}\right))</td>
</tr>
<tr>
<td>Kulczynski</td>
<td>(\frac{1}{2} \left(\frac{f_{11}}{f_{1+}} + \frac{f_{11}}{f_{1+}}\right))</td>
</tr>
<tr>
<td>Cosine Measure</td>
<td>(\frac{f_{11}}{\sqrt{f_{1+} \times f_{1+}}})</td>
</tr>
</tbody>
</table>

Their behavior differ in specific circumstances, for example with the total number of transactions involved in the computation, the ratio among the individual events or the ratio among the concurrence of the events and the total number of events.

These are not the only measures that can be used, there are plenty other measures in the literature such as: \(\phi\)/Correlation analysis, odds ratio, Kappa, Jaccard, mutual information, conviction, Gini index, certain factor, ... This is a current area of research and new measures can be defined for general or specific purposes. The properties that a measure of this kind must hold are:
6.5 APPLICATIONS

• If \( A \) and \( B \) are statistically independent then \( M(A, B) = 0 \)

• \( M(A, B) \) increases monotonically with \( P(A, B) \) when \( P(A) \) a \( P(B) \) remain unchanged

• \( M(A, B) \) decreases monotonically with \( P(A) \) (or \( P(B) \)) when \( P(A, B) \) and when \( P(B) \) (or \( P(A) \)) remain unchanged

Other properties that can also be desirable for these measures are:

• Symmetry \( (M(A, B) = M(B, A)) \)

• Invariant to row/column scaling

• Invariant to inversions \( (f_{11} \leftrightarrow f_{00}, f_{10} \leftrightarrow f_{01}) \)

• Null addition invariant (not affected if \( f_{00} \) is increased)

6.5 Applications

6.5.1 Discovering geographical associations from Twitter Data

The goal of this application is to discover associations among the geographical behavior of people inside a city using the events that they generate in Twitter. The dataset consists on tweets with a time stamp and longitude and latitude coordinates collected for several months (2.5 million tweets) in the Barcelona area \((30 Km \times 30 Km)\). We will perform a spatio-temporal analysis of the behavior of people in this geographical area, looking for patterns that show connections among different parts of the city.

One approach is to consider a tweet as an event performed by a user. The events of a user in a single day can be seen as a transaction assimilating the analysis to the one performed in market basket analysis. The attributes of a transaction in this case are the time and the position of the user. Now we can apply algorithms that discover association rules (frequent itemsets) to find connections among geographical positions inside the city.

Considering all possible values for position and time makes impossible to discover frequent patterns because it is very difficult to have items with the same valued. In order to obtain a suitable representation for frequent itemsets discovery we need to discretize the attributes. For time we can consider different meaningful hour groups (morning, evening, night, ...) or compute different groups of hours according to the statistical behavior of the temporal dimension. For the coordinates we can use an equally spaced grid (this raises the problem of what granularity to use) of we can use a clustering algorithm to divide the area on regions according to the different densities of the geographical positions (what clustering algorithm to use?).

For the analysis we decided to discretize time in intervals of 8 hours (0-7, 8-16, 13-23) and to use Two possibilities for coordinates:

• An equally spaced 300 \( \times \) 300 grid \((100 m \times 100 m)\)

• Clusters obtained by the leader algorithm \((radius 100 m)\)

The first option makes that a transaction has a determined number of attributes \((300 \times 300 \times 3)\), but no all the possibilities will appear. For second, the option the number of attributes will depend on the densities in the coordinates.

The steps followed for the analysis of the data were:
1. Generate the database of transactions using the discretizations decided for time and geographical position and grouping the daily event of the users

2. Discard the transactions that have only value for 1 attribute because they will not generate associations

3. Decide for a value for the support parameters, this needed for some experimentation, but some hints can be obtained from the densities of the discretizations

4. Apply FP-Grow because of the size of the dataset (hundreds of thousands of transactions) and the number of possible attributes (thousands of time-geographical position combinations)

5. Generate only the maximal itemsets to reduce the number or redundant patterns

Figure 6.4 show the results obtained using both geographical discretizations methods. The results are similar, but some additional patterns appear using the clustering algorithm for discretization because it is able to adapt better to the different densities of the geographical positions in the data. The grid is a too rigid method and can break densities in different cells of the grid reducing the actual count and precluding the discovery of the associations.

The interpretation of the results need some knowledge of the domain, but looking to the geographical positions that appear associated, most of them correspond to touristic places that are usually visited in the same day by people that visit Barcelona.
Figure 6.4: Frequent itemsets obtained from the twitter data using a grid discretization (top) and a clustering algorithm (bottom)
Structured Datasets

There are some domains where the patterns that can be discovered are more complex because instances are related to each other (not independent) or there are structured relations among the instance attributes. Mining these relationships is more interesting than obtaining patterns from the instances assuming there are no dependencies. For instance, we can have domains with a temporal component among all or groups of instances, relational databases (not a unique data table), or structured instances (trees, graphs).

In this chapter we will introduce from the unsupervised perspective methods for two kinds of structured data: sequences and graphs. We will talk about the same mining tasks that we have seen for unstructured data (clustering, frequent patterns) but applied to the different kinds of sequential data and data represented as graphs/trees.

7.1 Mining sequential data

Usually the methods applied for these data need to add specialized elements to be able to extract information from the sequential relationship. The assumption is that this relationship augments the semantic of the patterns that can be obtained from the individual instances.

Commonly, the relationship it is time, but any defined partial or total order among the instances can be used to define the sequence. Obviously, changing the relation can change the patterns that are discovered from the data. In this domain we can have a unique sequence, for instance, the remote connections to a computer, or a set of sequences, like the credit card purchases of a set of clients.

This kind of data can not usually be analyzed using classical techniques from time series analysis like autorregresive or moving average models (AR, MA, IMA) or other time series analysis methods like ARCH, GARCH or Kalman filter, because of the typology of the data.

These methods work for continuous data, but we will have to deal with:

- Multivariate qualitative data
- Very short series or a long series that has to be segmented in episodes
- Multivariate series where the interest is also in the relationships among the series
- Interest only in a part of the series (episodes, anomalies, novelty, ...)

There are multiple domains where we can find these kind data, for instance:

- Customer transactions over the time: Sets of transactions ordered by purchase time
• Web Data: Browsing activity of users in a web site
• Sensor data: History of events generated by a sensor (alarms, anomalies)
• Genome sequences: DNA/Protein sequences

The next sections will describe some methods for three different mining tasks for sequences, clustering of temporal series, discovery of frequent sequences and clustering of data streams. An excellent book covering different aspects of sequential data mining is [26].

7.1.1 Clustering for temporal series

To be able to apply clustering algorithms to temporal series we have to modify some elements of the process.

First, about the data, in some domains we will have a set of short continuous or discrete series, usually of the same length, but this is not a constraint. In other domains, we will not have directly the short sequences and we will have to segment the series to obtain the dataset. This leads of the problem of time series segmentation as a data preprocessing step.

There are different methods of segmentation. The simplest one is to extract fixed length windows from the series. Using information from the domain, we could decide what is the significant window length for our application. Using this value, the dataset is obtained by sliding the window along the series and extracting the segments. In this case we can decide to have overlapping or non overlapping windows, depending on the domain characteristics.

In some domains only specific segments of the series are significant, for instance, we have a series where some anomalous behavior appear from time to time. In this case, the segmentation preprocess appears as a problem of anomaly detection. Depending on the goal, perhaps we want to discover the relation among the normal and anomalous parts of the series, so the segmentation has to include both types of series. In this case, we can have an unbalanced dataset where the normal segments are over represented. This could be a problem for many clustering algorithms.

The second element that changes is how to represent the data. We could assume that each example is just a sequence of attributes and address the temporal dimension with other elements of the clustering process, for instance, with the distance function. But in the temporal domain there are many feature extraction methods that can be used to represent the series that can help to discover the patterns in the data. These methods can be used for dimensionality reduction complementary with the methods described in 1.3.1 and also for noise reduction.

For instance, there are methods that extract the extreme points (maximum, minima, inflection points) of the series that can be useful if these change points are the relevant ones for the behavior of the patterns. The original series is in the time domain, other methods extract features in the frequency domain by using a set of predefined shapes that can reconstruct the original signal, like Fast Fourier Transform (FFT) or wavelets (Haar, Daubechies, ...). Methods from Chaos theory are also used for transforming the series using combinations of different lags of the sequence. Finally, probabilistic methods can be used to model the series and the parameters of the models can be used as representation, for instance Hidden Markov Models or time series analysis methods like autorregresive models. Figure 7.1 shows some feature extraction methods applied to a time series like discrete fourier transform, discrete Haar wavelet transformation and 1-lag time transformation.

The third element that changes is how to measure the distance among the series. Even if we assume that all sequences are of the same length, the different speed or scale of the series can make that the usual distance functions (euclidean, hamming, ...) consider visually similar sequences as different. There are plenty distance functions specialized in time series that can be used to substitute the functions normally used by clustering algorithms. These distances
usually work even if the series have different lengths. Examples of these distances are, *Dynamic Time Warping* (DTW) that takes in account the different dynamic of two series looking for the best match among the data points using dynamic programming, *Longest Common Subsequence* (LCSS) that looks for the matching among two discrete series that obtains the longest math, *Edit Distance with Real Penalty* (ERP) that computes the minimum set of changes to transform a discrete sequence into another, or *Edit Distance on Real Sequence* (EDR) that extends the edit distance to real time series. Figure 7.2 shows the difference of using the euclidean distance or the DTW distance to compare two series. The DTW distance uses dynamic programming to find a match among the data points to maximize similarity.

### 7.1.2 Frequent sequences

In the previous section we assumed that we had a set of small discrete or continuous sequences, some times we are interested in the global sequence and we do not want to limit the length or the positions in the sequence of the patterns that we want to discover. The goal is to discover frequent sequence patterns inside the whole series, similarly to the patterns discovered by association rules.

The sequence does not need to have a temporal relationship, it can be, for instance, a string representing a DNA sequence. The values of each point of the sequence can be continuous/discrete/structured (transaction data) and form a univariate or multivariate sequence. There are different goals for the discovery process, including:

- Discovery of subseries that reveal some causality of events
- Discovery of abnormal/novel patterns (deviation from normal behavior)
- Discovery of frequent subpatterns
There are several methods that address some or all of these goals. We will introduce four different algorithms that tackle these problems using different approaches.

The first method is described in [57], it assumes a unique sequence of time discrete events and the goal of the discovery process is to find all frequent temporal sequences that describe the causal association of the events. This goal could be assimilated to the one of the association rule methods assuming a time order among the items of a transaction, so only the associations that follow the time order are possible. The goal is to discover if close events are causally related and the result is a set of frequent episodes of different length.

The algorithm extracts from the sequence all the possible windows of a specific length to build a dataset. The frequency of all the sequences is computed and the ones that have a count larger than a threshold are presented. This process is repeated for windows of increasing length until no more frequent sequences are found. Figure 7.3 shows an example of how this process works. In order to reduce the computational cost of the method, the same antimonotonic property of association rules is extended to sequences, so only the sequences that contain frequent sequences of shorter length are explored.
A second approach is the one presented in [48]. The goal is to discover the most frequent subsequences of a specific length from a continuous time series, this is called \textit{frequent motifs}. Being continuous data, it is very difficult that two subsequences are a exact match, so the algorithm assumes that there is a match if two sequences are at less than a specific distance.

The trivial algorithm to solve this task is $O(n^2)$, because all subsequences have to be compared to each other to determine their match. To reduce computational complexity, the windows are transformed to a simplified representation called Symbolic Aggregate Approximation (SAX). This representation discretizes the series, reducing its dimensionality with the property that the distance over this new representation is a lower bound of the distance in the original space, reducing the computational cost of the distance and the matching process.

The process to perform this transformation (see figure 7.4) chooses a length $l$ for the discretized sequence, and a number $s$ of symbols to discretize the continuous values. The values of the sequence are normalized using a Z-normalization. Then is divided in $l$ segments and the mean of each segment is used to determine the letter assigned to that segment. The values of a $\mathcal{N}(0,1)$ distribution are divided in $s$ equal sized intervals and the letter used for the discretization is the one that includes the mean value.

The process of finding the frequent patterns in a continuous time series is as follows. First, a window of a specified length is used to segment the sequence (overlapped windows). The windows transformed using SAX are stored in multiple hash tables using locality sensitive hashing. For determining the similar sequences, the sequences appearing in the same or contiguous buckets in different hash tables are compared. First, the distance in the transformed series is computed (cheap), if it is less than a threshold, then the exact distance is computed. If sequences are near, the sequences are stored as candidate frequent motifs. When all the sequences are processes, the candidates that appear more than a number of times are returned. This process reduces the computational cost to linear time, but at the cost of only an approximated solution to the problem.

For the third approach, described in [70], we have a sequence composed by discrete values from a vocabulary, basically a string. There is no assumption about the relationship among the elements of the sequence. The goal is to look for frequent patterns of any length using different constraints. The pattern can be exact, it can have some equivalence classes among the symbols, so they can be counted as the same, or there can be wildcards, so we can have holes in the patterns where the symbols that appear are not important.

These algorithms use specialized data structures to discover the patterns within a reasonable computational cost, usually suffix-trees or related structures. The construction of these structures are sometimes a preprocess (later we extract the patterns) or given a threshold frequency we obtain only the patterns we are looking for by building the data structure.

We are going only to describe the algorithm for obtaining exact frequent patterns from a sequence. The first step is to obtain a suffix-trie from the sequence. A given ($k$) parameter sets...
the minimum frequency of the patterns that have to be extracted. The structure begins with
the empty string, then processes the sequence as follows:

1. For each character that belongs to the alphabet of the sequence

   (a) Compute a list with the next position to the occurrence of the character

   (b) If the character appears more than $k$ times we create a new descendant in the trie

2. Recursive call for each descendant created

The following example shows how the frequent sequences from the sequence ABACABBAACA
are obtained with frequency at least 2. The first level of the tree counts how many frequent
sequences of length 1 appear, each symbol stores the list of the next positions where the symbol
appears in the sequence. In this case $A$ appears 6 times, $B$ appears 3 times and $C$ appears 3
times, so all sequences are frequent.

Now each branch can be process in parallel looking for symbols that can continue each one.
In this case the symbol $A$ can only be cotinued by the symbols $B$ and $C$ that have count two.
The symbol $B$ only can be continued by $A$ and the symbol $C$ only can be continued by $A$.

Again we can process all the possible continuations of the branches in parallel. In this case
we find that only the sequence $AC$ has the symbol $A$ as a continuation with frequency 2.
Finally, in chapter 6 we talked about how to discover frequent patterns in a database of transactions, but we considered that these transactions were independent. Sometimes we have sequences of transactions, that is, each element of the sequence is a set of elements chosen from a list of possible events. For instance, we can consider all the purchases of a client in a supermarket a sequence where each purchase contains the list of item that have been bought. We can be interested on what is common/frequent in the temporal behavior of the clients of the supermarket. This could for example allow answering what is going to buy a client if it has a behavior that is contained in a frequent sequence of purchases.

For this task, the definition of sequence has to be adapted:

- Given an ordered list of elements:

\[ s = \langle e_1 e_2 e_3 \ldots \rangle \]
• Each element contains a collection of items

\[ e_i = \{i_1, i_2, ..., i_k\} \]

Also, the definition of subsequence has to be redefined:

• A sequence \(< a_1, a_2, ..., a_n >\) is contained in another sequence \(< b_1, b_2, ..., b_m >\) (\(m \geq n\)) if exists integers \(i_1 < i_2 < ...i_n\) such that \(a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, a_n \subseteq b_m\)

The algorithm GSP (Generalized Sequential Patterns) presented in [5, 67] is an apriori-like approach to the mining of sequential patterns. A similar anti-monotonic property exists among a sequence and its subsequences:

• Given a sequential pattern \(s\) all its subsequences are also sequential patterns

• Given a sequential pattern \(s\) if \(s'\) is a subsequence then \(\text{support}(s) \leq \text{support}(s')\)

This means that generating all the frequent sequences for a database can be done using a sequential approach by enumerating all the candidates to frequent sequences and pruning the ones that contain non frequent sequences.

The algorithm, for each iteration generates the sequences of length \(k + 1\) by merging the frequent sequences of length \(k\). For the generation of candidates all items in each transaction are listed alphabetically to avoid generating repeated sequences. Each iteration the database is scanned to test for the candidates and compute their support. The sequences that are not frequent are pruned from the list of candidates.

The method for candidate generation is more complex than in the association rules case. Given \(s_1\) and \(s_2\), frequent sequences of length \(k\) that are identical except for the last element, there are three possible cases that have to be contemplated:

• If the last element of \(s_1\) and \(s_2\) contain only one item. Assuming that \(s_1 = sx\) and \(s_2 = sy\) where \(s\) is the maximum common prefix of \(s_1\) and \(s_2\) and \(x, y\) two items, then are candidates of length \(k + 1\) \(s(xy), sxy\) and \(syx\)

\[(ab)(ac)ab + (ab)(ac)ac \Rightarrow (ab)(ac)a(bc), (ab)(ac)abc, (ab)(ac)acb\]

• If the last element contains more than one item, and in alphabetical order these last item-sets are identical except for the last item. Assuming that \(s_1 = s(x_1 \cdots x_{m-1}x_m)\) and \(s_2 = s(x_1 \cdots x_{m-1}x_{m+1})\) where \(s\) is the maximum common prefix of \(s_1\) and \(s_2\), then is a candidate \(s(x_1 \cdots x_{m-1}x_mx_{m+1})\)

\[(ab)(ac)(ab) + (ab)(ac)(ac) \Rightarrow (ab)(ac)(abc)\]

• If the last element of \(s_2\) contains one item, and the second last element of \(s_2\) is identical to the last element of \(s_1\) except for one item that is the last in the last element of \(s_1\) in alphabetical order. Assuming that \(s_1 = s(x_1 \cdots x_{m-1}x_m)\) and \(s_2 = s(x_1 \cdots x_{m-1}y)\) where \(s\) is the maximum common prefix of \(s_1\) and \(s_2\), then is a candidate \(s(x_1 \cdots x_{m-1}x_mx_{m+1})y\)

\[(ab)(abc) + (ab)(ab)a \Rightarrow (ab)(abc)a\]
7.1.3 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, the algorithm described in [2] introduces 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 microclusters using an incremental clustering algorithm similar to the leader algorithm. When new data arrives, it is incorporated to an existing microcluster if it is similar enough, or generates a new microcluster. The number of microclusters 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 microclusters is fixed, so they are merged to maintain this number and in this way to evolve with the changes in the data. Periodically the microclusters are stored to be able to obtain historical snapshots of the data stream.

The off-line phase uses the microclusters stored for a window of time. These are merged to obtain a set of microclusters 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.

A similar approach is described in [20]. 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 weight fades exponentially with time, that is known as time damping model. These weighted examples are classified according to their surrounding density as core-microclusters, 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 clusters 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 7.5 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.

7.1.4 Application: Sequential data

This application uses signals recorded using a set of sensors over the lumbar vertebrae of an anesthetized cat (from L4 to L7), see figure 7.6. The goal is to determine the effect of different experimental conditions related to stimulation of nociceptive receptors of the cat’s limbs. The object of study are the peaks generated spontaneously by groups of neurons in these vertebrae and the synchronizations of these peaks in the different recording points. From these synchronizations, the frequent sequential patterns can be extracted to detect changes in behavior.

The spontaneous neuron activations are only a part of the whole signal, so previous to the analysis the segments of the signal that correspond to these activations have to be extracted. For doing this, a specialized peak finding algorithm can be used, so, given a window length, only the parts of the signal that have a peak in the center are extracted.

This peaks will be the dataset to analyze for studying the behaviour of the signal. A preprocessing step can be also performed to reduce the noise of the signal (see figure 7.7). The model of the peaks behavior will be obtained using a clustering algorithm, in this case the euclidean distance will be used to compare the peaks. The clustering algorithm used is K-means, determining the optimal number of clusters using a stability clustering method. This clustering results in a tentative dictionary of peaks shapes (see figure 7.8) that can be used to
Figure 7.6: Recording of activity of neurons in the lumber segments of the spinal cord

Figure 7.7: Peaks in the signal due to spontaneous neuron activations
discretize the signal in a sequence of symbols.

A first goal could be to study if there are specific sequences of peaks in the individual signals that appear frequently. To obtain these frequent sequences, the clusters of the peaks can be used to transform each signal to a string where each symbol is the cluster of the peak. In this sequence, an algorithm that extracts all frequent sequences given a threshold can be applied. Different experimental conditions could change the sequences that are frequent.

A second goal is to study how these peaks synchronize in the different segments of the lumbar vertebrae, so using the time of the sequences of peaks, a synchronization finding algorithm is used to determine what peaks from the different signals appear inside a predefined window of time (see figure 7.9). We can view these synchronizations as transactions, where each one has a subset of the possible synchronizations as represented in figure 7.10. Each column represents the segments at left and right and the lumbar segments from L4 to L7, the red squares represent that a peak at that position have occurred.

In order to create the transactions sequence, all the transactions that are at time distance less that a specific threshold are considered corresponding to the same sequence. This allows building a database of sequential transactions, that can be processed using a frequent sequence algorithm. The result would be frequent concurrent activations of neurons that can have different behavior depending on the experimental conditions.

### 7.2 Graph mining

There is a lot of information that has a relational structure and can be represented as graphs or trees. Figure 7.11 shows some examples such as social networks, relational databases, XML documents or chemical molecules. Methods and models applied to unstructured data are not expressive enough to discover interesting patterns. Sometimes the structure can be flattened, but lots of interesting information are lost in the process. For instance, a relational database can be merged in a unique table, but the links among the records of different table will disappear or will be represented by attributes indicating that certain attribute are not applicable for specific examples. Graph data can also be linearized as strings using graph traversal algorithms and treated as strings or sequences.

Historically we can find different approaches to the discovery of patterns in graphs/trees, such as Inductive logic programming (ILP) where the structure is represented using logic formulas, or different graph algorithms, like for instance classic algorithms for detecting dense subgraphs (cliques), graph isomorphism algorithms or graph partitioning algorithms.

Something important to note is that most of the problems used to discover structure in graphs are NP-Hard, like graph partitioning (not for bi-partitioning) or graph isomorphism. This means that most of the algorithms in this area resort to approximate solutions for these problems.

The next sections will describe three possible mining goals in graph structures, the partition of one large graph in a set of dense graphs, the cluster of datasets where the examples are graphs and the discovery of frequent structures in a database of graphs.

An excellent book that covers all these topics is [4].

#### 7.2.1 Graph partitioning

Some information can be described as a large graph where several instances are connected by different types of relations, for instance a social network or web pages. In these data we are interested in discovering interesting substructures by dividing the graph in subgraphs (k-way partitioning, node clustering) or by extracting dense substructures that represent significant parts of the graph.
Figure 7.8: Clustering of the peak

Figure 7.9: Synchronization of peaks from different lumbar vertebrae

Figure 7.10: Transactions corresponding to synchronizations of different lumbar vertebrae
The simplest partitioning of a graph is to divide the graph in two subgraphs. We assume that edges have values as labels (similarity, distance, ...) that can be used to compute the partition. This problem is known as the minimum cut-problem:

“Given a graph, divide the set of nodes in two groups so the cost of the edges connecting the nodes between the groups is minimum”

This problem is related to the maximum flow problem that can be solved in polynomial time. The general problem (more than two clusters) is NP-hard. This is not news, because we already knew that clustering is NP-hard and partitioning a graph is the exact same problem.

Graph partitioning for more than two clusters can be solved approximately by local search algorithms (hill climbing), the simplest one is the Kerninghan-Lin algorithm, it is as follows:

1. Start with a random cut of the graph
2. Interchange a pair nodes from different partitions that reduces the cut
3. Iterate until convergence
There are different variations of this algorithm that changes the strategy for selecting the pair of nodes to interchange. An extensive survey of graph partitioning algorithms from the graph theoretical perspective can be found in [31].

Being this the same problem as clustering, their algorithms can be adapted to obtain a graph partition. For instance, K-means and K-medoids can be adapted so the nodes of the graph act as prototypes and the objective functions used to define the node membership to clusters can be computed using the geodesic distance (shortest path) instead of the euclidean distance.

As we saw in section 2.3.4, there are several clustering algorithms that are based on the distance graph. For instance, spectral clustering can be applied using the distance matrix computed with the geodesic distance, or other appropriate distance computed from the values of the edges of the graph and define the Laplacian matrix. Performing the eigendecomposition and finding the largest eigenvalues determine the number of clusters and how to assign the nodes to the clusters. A general survey about graph clustering can be found in [64].

The area of social networks has several algorithms to perform the partition of a large graph. The most known is the Girvan-Newman Algorithm that is based on the concept of edge betweenness centrality:

\[ B(e) = \frac{\text{NumConstrainedPaths}(e, i, j)}{\text{NumShortPaths}(i, j)} \]

Because it is costly to compute this measure for all pairs of nodes, there are variants that are based on the Random Walk Betweenness that computes how often a random walk starting on node \( i \) passes through node \( j \). This algorithm is also connected with the well known Page Rank algorithm (the one that uses Google search). These measures detect bridges among dense components, that is edges that are not in the shortest paths between pairs of nodes, that can be used as criterion to delete edges from the graph.

The algorithm is as follows:

1. Rank edges by \( B(e) \)
2. Delete edge with the highest score
3. Iterate until a specific criteria holds (eg. number of components)

Not always a complete partitioning of the graph is necessary, we could be interested only on dense subgraphs that can represent interesting behaviors. The idea is that the rest of the graph is not dense enough to be supported by the data. There are different types of (pseudo)dense subgraphs that can be computed, using also different goals, like having a minimum size, having the best or a minimum rank according a quality measure or being or not overlapped. The common types of dense graphs are:

- Clique: All nodes are connected
- Quasi-clique: Almost all nodes are connected, so a minimum density or minimum degree has to be given
- K-core: Every node connects to at least k nodes, so there is a minimum density for all nodes
- K-plex: Each node is missing no more than k-1 edges, the same but from the complementary perspective
There are different approaches to find these graphs. The first one is exact enumeration, that involves in the worst case to generate all the possible subsets of vertices and to check if they form a specific type of dense graph. This is computationally unfeasible (NP-Complete) for most of the types of dense graphs except for K-cores that can be computed on polynomial time.

The second possibility is to use a heuristic approach like Shingling ([36]), that involves the use of hash functions to identify vertices with common neighbors or the GRASP Algorithm ([1]) that uses local randomized search to detect quasi-cliques.

7.2.2 Clustering of graphs/trees

We are used to examples that are described by attributes that do not have explicit relations among them. Some data must be described as a graph/tree to be able to capture all their characteristics. In this case the relations itself can have values, for instance XML documents or chemical molecules. To partition these data in clusters it is more complex because the notion of distance among the examples and of cluster prototype have to be adapted.

There are some approaches to this problem that embed the set of graphs in a metric space by obtaining a vector representation of the data, so the usual clustering algorithms can be used. For instance, using the spectral properties of the graphs, the Laplacian matrix of the graphs can be transformed, so similar graphs will have shorter distances in this representation.

Also, kernel functions specific for graphs can be used to measure similarity like for instance diffusion or convolution kernels, or based on random walks. The distance matrix obtained using these kernels can be used for hierarchical clustering or it can be transformed to a vector representation using some of the dimensionality reduction methods explained in section 1.3.2, like for instance MDS or ISOMAP. The main problem of these approaches is that there is no meaningful prototype that can be used for representing the characteristics of the graphs from a cluster.

The XProj algorithm ([3]) uses a different approach to be able to maintain information about the common characteristics of the examples in a group. This algorithm partitions a set of XML documents (we have trees in this case) in K classes using a strategy similar to K-means (see algorithm 7.1). The summarization and similarity are based on frequent substructures that the trees in a cluster have. For each partition, the frequent substructures of size $l$ are computed and used as prototype. The number of substructures of the prototypes in a tree is used as similarity function. The main problem with this algorithm is the computational cost of computing the common substructures that is reduced by linearizing the trees using a pre-order traversal algorithm.

7.2.3 Frequent subgraphs

Other approach to obtain patterns from sets of graph/trees is to search for their frequent substructures. These can give information about the correlation/causal relationship among the different elements that form the graphs as with frequent itemsets algorithms.

It is difficult to perform the task directly in the graph representation. Also, it involves the problem of graph isomorphism or finding if a graph is contained other that are computationally expensive.

To tackle this problem, the structures are usually transformed to some kind of canonical representation, such as the adjacency matrix or a graph traversal. These representations give a unique code for each different graph. Patterns can be discovered from these codes that will be related to the patterns that actually appear in the original graphs. Some of these approaches use as pruning method to reduce the cost of the search the same anti-monotonic properties and
Algorithm 7.1 XProj graph clustering algorithm

**Algorithm:** XProj

Partition the documents in K random subsets
Compute the prototypes (frequent substructures length $l$) $S_k$

**repeat**

- Assign each document to the most similar prototype ($S_k$)
  
  /* let $M_1, \ldots, M_k$ be the new partitions */

**foreach** $M_i$ **do**

- Recompute the most frequent substructures of length $l$ of $M_i$
  
  **if** frequency of substructures > $\min_{\text{sup}}$ **then**
    
    Modify the prototype
  
- **end**

**end**

until convergence criteria

---

strategies used in association rules, beginning with the simplest patterns and increasing their size by adding a new element at a time until no new frequent patterns are found.

An example of this strategy is the algorithm gSpan ([75]). Its goal is to discover frequent patterns in datasets of graphs that can have labels in their edges and their vertices. A canonical representation is used to reduce the cost to compute graph isomorphism. This representation is based on the tree obtained by the depth first traversal of the graph and forcing a lexicographical order among labels. This representation transforms a graph into a string that contains all the labels of the graph. With this representation an ordering is induced over all the graphs that can be used to explore all possible subgraphs.

Many depth first search (DFS) traversals of a graph can obtained. We assume an initial vertex ($v_0$) and an order among vertices ($i = 0 \ldots n$). For any DFS traversal of a graph, we define the forward edges (those with $i < j$) and the backward edges (those with $i > j$). We define also a linear order among edges, given $e_1 = (i_1, j_1)$ and $e_2 = (i_2, j_2)$:

- if $i_1 = i_2$ and $j_1 < j_2$ then $e_1 \prec T e_2$
- if $i_1 < j_1$ and $j_1 = i_2$ then $e_1 \prec T e_2$
- if $e_1 \prec T e_2$ and $e_2 \prec T e_3$ then $e_1 \prec T e_3$

Given a DFS tree for a graph, an edge sequence can be defined based on the order relation $\prec_T$, this will be the DFS code of the graph. Given all the DFS codes for a graph, a linear lexicographic order among codes can be defined using the linear order among edges and the linear order for its labels $\prec_L$. The Minimum DFS code using that order relations is the **canonical label** for $G$.

Figure 7.12 shows an example of code for a graph. A DFS has to enumerate all the edges of a graph, following vertices in depth first order. Given the graph on the left, many DFS can be obtained, if we determine a specific order for the vertices (right graph) the DFS code shown on the right of the figure is obtained.

Using these codes for all the graphs in the dataset, the gSpan algorithm generates all the frequent subgraphs that have a support larger than a specified threshold.

It has an initialization step that eliminates of the graph all the edges and vertices that have a support less than the threshold. This is so because of the anti-monotonic property of support (if they are no frequent cannot from part of frequent subgraphs), then the process
begins with the DFS codes of all the graphs consisting of one edge. These graphs are explored using the lexicographic order to avoid repeating graphs as presented in algorithm 7.2. The main algorithm of gSpan receives a DFS code, checks if it is a minimal code and if it is so, scans the database for all possible extensions of the code following the DFS order (rightmost extensions) that are frequent, adding them to a candidates list. Finally, it recursively applies gSpan in lexicographic order to the candidates list. This is detailed in algorithm 7.3.

In figure 7.13 an example of the algorithm applied to a graph database is depicted. The graph database consists of two graphs, the algorithm looks for all frequent subgraphs that have a support larger than two. Labels are omitted in the graphs to simplify the example and a vertices order \([\text{yellow} < \text{green} < \text{red}]\) and right to left is assumed. The algorithm begins with all the one edge graphs that have a count larger or equal than 2, in this case \(\text{yellow} – \text{yellow}, \text{yellow} – \text{green} \text{ yellow} – \text{red}\). Following the lexicographical order, all possible extensions of the \(\text{yellow} – \text{yellow}\) graph are explored, in this case four possibilities, only two of them are different. The graphs are \(\text{yellow} – \text{yellow} – \text{green}\) and \(\text{yellow} – \text{yellow} – \text{red}\). The repeated extensions are detected because their code is not minimal. Being frequent, both graphs are explored using the lexicographic order. The algorithm continues until all the different frequent subgraphs are enumerated as can be seen in figure 7.13.

There are other algorithms that target specific graphs or trees in order to take advantage of their specific properties to reduce the computational cost of the search but with the cost of reducing the type of structures that can be discovered. For instance, the FreeTreeMiner
Algorithm 7.3 Core of the gSpan algorithm

Algorithm: gSpan($s$, $D$, $MinSup$, $S$)

if $s \neq mincode(s)$ then return
Insert $s$ into $S$
$C = \emptyset$
Scan $D$
begin
  Find every edge $e$ such that $s$ can be right-most extended to frequent $s \ast e$
  Insert $s \ast e$ into $C$
end
Sort $C$ in DFS lexicographic order
foreach $s \ast e$ in $C$ do
  gSpan($s \ast e$, $D$, $MinSup$, $S$)
end

algorithm described in [62] discovers frequent free trees inside graph databases. A free tree is a connected acyclic labeled graph (can be seen as a rootless tree). The algorithm also defines a canonical representation for a tree, computed using as root of the tree the most centered node and establishing a lexicographic order for its labels. A level traversal of the tree defines the linearized representation. This representation aims to reduce the problem of graph isomorphism. The search is performed level wise starting with a free tree with only a node and extending each iteration the candidate trees with a node that follows the canonical order. Figure 7.14 shows an example.
Figure 7.13: Example of gSpan applied to a small graph database

Figure 7.14: Example of the FreeTreeMiner Algorithm


