Advanced Human Language Technologies

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Outline

Assignment

Write a python program that parses all XML files in the folder given as argument and recognizes and classifies sentences stating drug-drug interactions. The program must use a neural network approach.

```
$ python3 ./nn-DDI.py data/Devel/
DDI-DrugBank.d398.s0|DDI-DrugBank.d398.s0.e0|DDI-DrugBank.d398.s0.e1|effect
DDI-DrugBank.d398.s0|DDI-DrugBank.d398.s0.e0|DDI-DrugBank.d398.s0.e2|effect
DDI-DrugBank.d211.s2|DDI-DrugBank.d211.s2.e0|DDI-DrugBank.d211.s2.e5|mechanism
...
```

Outline

General Structure

The general structure is basically the same than for the traditional ML approach:

- Two programs: one learner and one classifier.
- The learner loads the training (Train) and validation (Devel) data, formats/encodes it appropiately, and feeds it to the model, toghether with the ground truth.
- The classifier loads the test data, formats/encodes it in the same way that was used in training, and feeds it to the model to get a prediction.

In the case of NN, we don't need to extract features (though we do need some encoding)

Input Encoding

- The input/output layers of a NN are vectors of neurons, each set to 0/1.
- Modern deep learning libraries handle this in the form of *indexes* (i.e. just provided the *position* of active neurons, ommitting zeros).
- For instance, in a LSTM, each input word in the sequence may be encoded as the concatenation of different vectors each containing information about some aspect of the word (form, lemma, PoS, suffix...)
- Each vector will have only one active neuron, indicated by its *index*. This input is usually fed to an embedding layer.
- Our learned will need to create and store *index* dictionaries to be able to intepret the model later. See class *Codemaps* below.

Outline

Outline

Learner - Main program


```python
def learn(traindir, validationdir, n_epochs, modelname) :
   ## learns a NN model using trainfle as training data, and validationfile
   ## as validation data. Saves learnt model in a file named modelname

   # load train and validation data
   traindata = Dataset(trainfile)
   valdata = Dataset(validationfile)
   # create indexes from training data
   max_len = 150
   codes = Codemaps(traindata, max_len)
   # encode datasets and load them in a Data loader
   train_loader = encode_dataset(traindata, codes)
   val_loader = encode_dataset(valdata, codes)
   # build network
   network = ddiCNN(codes)
   optimizer = optim.Adam(network.parameters())
   # perform training, computing validation stats at each epoch
   for epoch in range(n_epochs):
      train(epoch)
      validation()
   # save model and indexs
   os.makedirs(modelname,exist_ok=True)
   torch.save(network, modelname+"/network.nn")
   codes.save(modelname+"/codemaps")
```

Outline

Classifier - Main program


```
def predict(modelname, datadir) :
   '''
   Loads a NN model from file 'modelname' and uses it to extract drugs
   in datafile. Output results to stdout.
   '''

   # load model and associated encoding data
   network = torch.load(modelname+"/network.nn")
   network.eval()
   codes = Codemaps(modelname+"/codemaps")

   # encode datasets and load it in a Data loader
   testdata = Dataset(datadir)
   test_loader = encode_dataset(testdata,codes)

   Y = []
   # run each validation example and report validation loss
   for X in test_loader:
      # X is a list of input tensors (no labels were loaded in the
     dataloader)
      y = network.forward(*X) # run example through the network
      # add results to result list
      Y.extend([codes.idx2label(torch.argmax(s)) for s in y])

   # extract relations from result list
   output_interactions(testdata, Y)
```

Note: Observe the output structure (one class per sentence+pair), different from the NER task (one class per token).

Outline

Auxiliary classes - parse_data

Processing de whole dataset with StanfordCore takes a long time, so it is convenient to run it once and for all:

```

# preprocess a dataset with StanfordCore, and store result in a
# pickle file for later use.
# usage:  ./parse_data.py data-folder filename
#   e.g.  ./parse_data.py ../../data/train train

datadir = sys.argv[1]
filename =  sys.argv[2]

data = Dataset(datadir)
data.save(filename)
```

Auxiliary classes - Dataset

```
class Dataset:
   ## constructor:
   ## If 'filename' is a directory, parses all XML files in datadir,
     tokenizes
   ## each sentence, and stores a list of sentence/pairs, each
   ## of them as a parsed tree with masked target entities
   ## tokens (word, start, end, gold_label), plus associate ground truth.
   ## If 'filename' is a '.pck' file, load data set pickle file
   def __init__(self, filename)

   ## saves dataset to a piclke file (to avoid repeating parsing)
   def save(self, filename)

   ## iterator to get sentences in the dataset
   def sentence(self)
  '''
```

Class `Dataset` will *mask* the target entities in the input sentence:

Original sentence: *Exposure to oral ketamine is unaffected by itraconazole compounds but greatly increased by ticlopidine.*

Pair	Masked sentence
e0-e1	Exposure to oral DRUG1 is unaffected by DRUG2 but greatly increased by DRUG_OTHER.
e0-e2	Exposure to oral DRUG1 is unaffected by DRUG_OTHER but greatly increased by DRUG2.
e1-e2	Exposure to oral DRUG_OTHER is unaffected by DRUG1 but greatly increased by DRUG2.

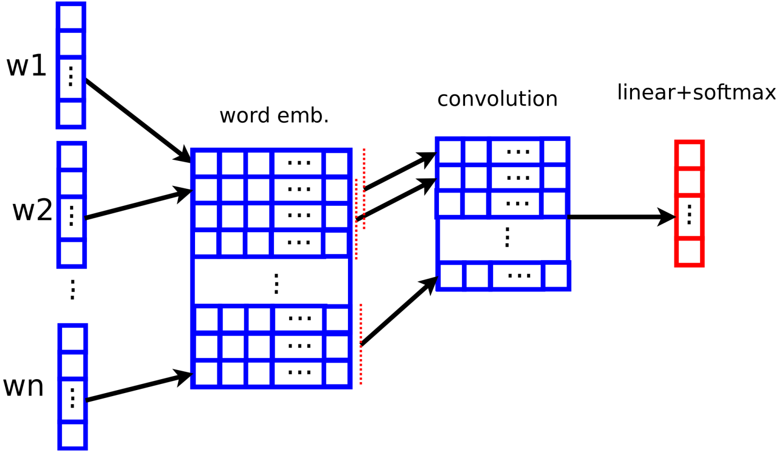
Auxiliary classes - Codemaps

```python
class Codemaps :
    # Constructor: create code mapper either from training data, or
    #              loading codemaps from given file.
    #              If 'data' is a Dataset, and lengths are not None,
    #              create maps from given data.
    #              If data is a string (file name), load maps from file.
    def __init__(self, data, maxlen=None, suflen=None)
    # Save created codemaps in file named 'name'
    def save(self, name)
    # Convert a Dataset into lists of word codes and sufix codes
    # Adds padding and unknown word codes.
    def encode_words(self, data)
    # Convert the gold labels in given Dataset into a list of label codes.
    # Adds padding
    def encode_labels(self, data)
    # get word index size
    def get_n_words(self)
    # get suf index size
    def get_n_sufs(self)
    # get label index size
    def get_n_labels(self)
    # get index for given word
    def word2idx(self, w)
    # get index for given suffix
    def suff2idx(self, s)
    # get index for given label
    def label2idx(self, l)
    # get label name for given index
    def idx2label(self, i)
```

Required functions - network.py


```python
class ddiCNN(nn.Module):

    def __init__(self, codes) :
        super(ddiCNN, self).__init__()

        # get sizes
        n_words = codes.get_n_words()
        n_labels = codes.get_n_labels()
        max_len = codes.maxlen

        # create layers
        self.embW = nn.Embedding(n_words, 100, padding_idx=0)
        self.cnn = nn.Conv1d(100, 32, kernel_size=3, stride=1, padding='same')
        self.out = nn.Linear(32*max_len, n_labels)

    def forward(self, w):
        # run layers on given data
        x = self.embW(w)
        x = x.permute(0,2,1)
        x = self.cnn(x)
        x = func.relu(x)
        x = x.flatten(start_dim=1)
        x = self.out(x)
        return x
```

Network architecture



Outline

Build a good NN-based DDI detector

- DDI is not a sequence tagging task (which assign one label per word), but a sentence classification, where a single label is assigned to the whole sentence (or sentence + entity pair in this case).
- Good results may be achieved using a CNN, as in the provided example.
- The problem also may be approached with an LSTM. Note that instead of getting the output at each word, only the output at the end of the sequence must be used (or the output of all words must be combined to feed further layers).
- It is also possible to combine LSTM and CNN layers.
- You will need to add one Embedding layer after the input, that is where the created indexes will become handy.
- You may get inspiration for an architecture from these examples: [1], [2],[3],[4], some of the papers provided in labAHLT package in `papers/SharedTask/otherSystems`, or just googling for *semeval DDI neural networks*.

Build a good NN-based DDI detector

Strategy: Experiment with different NN architectures and possibilities.
Some elements you can play with:

- Embedding dimensions, number and kind of layers, used optimizer...
- Using just CNN, just a LSTM, or a LSTM+CNN combination
- Using lowercased and/or non lowercased word embeddings
- Initialitzing embeddings with available pretrained model
- Using extra input (e.g. lemma embeddings, PoS embeddings, suffix/prefix embbedings, ...)
- Adding extra dense layers, with different activation functions
- Using pretrained transformers such as Bert as the first layers of your network.
- Adding attention layers
- ...etc.

Build a good NN-based DDI detector

Warnings:

- Neural Network training uses randomization, so different runs of the same program will produce different results. For repeatable results, use a random seed (and/or run the training several times).
- During training, *accuracy* on training and validation sets is reported. Those values are usually over 85%. However, this is due to the fact that most of the words have label "O" (non-drug). Accuracy values around 85% correspond to very low F_1 values. To get a reasonable F_1, validation set accuracy should reach about 89-90%.

  To precisely evaluate how your model is doing, do not rely on reported accuracy: run the classifier on the development set and use the evaluator.

Outline

Exercise Goals

What you should do:

- Work on your architecture and input vectors. It is the component of the process where you have most control.
- Experiment with different architectures and hyperparameters.
- Experiment with different input information
- Keep track of tried variants and parameter combinations.

What you should **NOT** do:

- Alter the suggested code structure (i.e. change only `network.py` and `Codemaps`).
- Produce an overfitted model: If performance on the test dataset is much lower than on devel dataset, you probably are overfitting your model.

Exercise Goals

Orientative results:

- Provided CNN architecture gets a macroaverage F1 about 50%. Input information includes only embeddings for word forms.
- The NN may be extended with extra input information, additional convolutional layers (either separate for each input or after concatenating them), changing their size, changing the size/stride of the convolutional kernel, adding LSTM layers (before the CNN, after it, or instead of it), maxpool layers (typically after the CNN or LSTM), etc.
- With appropriate improvements, macro average F1 can be raised to over 60%.

Deliverables

Write a report describing the work carried out in both NN exercises
The report must be a **single self-contained PDF document**, under ~10 pages, containing:

- *Introduction:* Context of the report.
- *NN-based NERC*
  - *Architecture:* What architectures did you try, and which was finally selected.
  - *Input information:* What input data did you use, and how did you encode it to feed the NN. Explain changes made on class `Codemaps`.
  - *Experiments:* Results obtained on the **devel** dataset for tested combinations of input layers, architectures, and hyperparameter settings. No need to report the whole result table for all experiments, a summary is enough.
  - *Final results:* Complete result table for best performing models on **devel** dataset. Result table of their application to **test** dataset.
  - *Code:* Include your `network.py`. **Do not include any other code.**

Deliverables (continued)

- *NN-based DDI*
  - *Architecture:* What architectures did you try, and which was finally selected.
  - *Input information:* What input data did you use, and how did you encode it to feed the NN. Explain changes made on class `Codemaps`.
  - *Experiments:* Results obtained on the **devel** dataset for tested combinations of input layers, architectures, and hyperparameter settings. No need to report the whole result table for all experiments, a summary is enough.
  - *Final results:* Complete result table for best performing models on **devel** dataset. Result table of their application to **test** dataset.
  - *Code:* Include your `network.py`. **Do not include any other code.**
- *Conclusions:* Final remarks and insights gained using NN for NERC and relation extraction. Comparison with other ML approaches used during the course in terms of development cost and achieved performance.

Keep result tables in your report in the format produced by the evaluator module. Do not reorganize/summarize/reformat the tables or their content.