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(trainfile, validationfile, 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, suf_len)

   # build network
   model = build_network(codes)
   with redirect_stdout(sys.stderr) :
      model.summary()

   # encode datasets
   Xt = codes.encode_words(traindata)
   Yt = codes.encode_labels(traindata)
   Xv = codes.encode_words(valdata)
   Yv = codes.encode_labels(valdata)

   # train model
   with redirect_stdout(sys.stderr) :
   model.fit(Xt, Yt, batch_size=32, epochs=10, validation_data=(Xv,Yv),
     verbose=1)

   # save model and indexs
   model.save(modelname)
   codes.save(modelname)
```

Outline

Classifier - Main program

```python
def predict(modelname, datafile, outfile) :
    '''
    Loads a NN model from file 'modelname' and uses it to extract drugs
    in datafile. Saves results to 'outfile' in the appropriate format.
    '''

    # load model and associated encoding data
    model = load_model(fname)
    codes = Codemaps(fname)

    # load and encode data to annotate
    testdata = Dataset(datafile)
    X = codes.encode_words(testdata)

    # tag sentences in dataset
    Y = model.predict(X)
    Y = [codes.idx2label(np.argmax(s)) for s in Y]

    # extract relations
    output_interactions(testdata, Y, outfile)
```

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

```
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 - build_network

```
def build_network(codes) :

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

    # word input layer & embeddings
    inptW = Input(shape=(max_len,))
    embW = Embedding(input_dim=n_words, output_dim=100,
                     input_length=max_len, mask_zero=False)(inptW)

    conv = Conv1D(filters=30, kernel_size=2, strides=1,
                  activation='relu', padding='same')(embW)
    flat = Flatten()(conv)

    out = Dense(n_labels, activation='softmax')(flat)

    model = Model(inptW, out)
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=['accuracy'])

    return model
```

Required functions - build_network

- 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 (`return_sequences=True`), only the output at the end of the sequence must be used (`return_sequences=False`).
- 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*.

Outline

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. Run the same model several times.
- During training, Keras reports accuracy on training set and on validation set. Those values are usually over 85%. However, this is due to the fact that about 85% of the pairs have interaction "`null`" (no-interaction). Thus, 85% accuracies correspond very low F_1 values. To get a reasonable F_1, val_accuracy must 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.
- Produce an overfitted model: If performance on the test dataset is much lower than on devel dataset, you probably are overfitting your model. Note that a very large/deep NN applied to a reduced dataset (as is this case) will just *remember* the training data and thus will not generalize properly.

Exercise Goals

Orientative results:

- Provided CNN architecture gets a macroaverage F1 over 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.
- Goal: achieve ∼65% macro average F1.

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:* What is this report about. What is the goal of the presented work.
- *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.
  - *Code:* Include your `build_network` function (and any other function it may call), properly formatted and commented. **Do not include any other code.**.
  - *Experiments and results:* Results obtained on the **devel** and **test** datasets, for different architecture/hyperparameter combinations you deem relevant.

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.
  - *Code:* Include your `build_network` function (and any other function it may call), properly formatted and commented. **Do not include any other code.**.
  - *Experiments and results:* Results obtained on the **devel** and **test** datasets, for different architecture/hyperparameter combinations you deem relevant.
- *Conclusions:* Final remarks and insights gained in this task.

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