Advanced Human Language Technologies

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Outline

Task 2.2 - DDI using neural networks

Assignment

Write a python program that parses given XML file and recognizes and classifies sentences stating drug-drug interactions. The program must use a neural network approach.

```
$ python3 ./nn-DDI.py devel.xml result.out
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 (*one-hot encoding*), 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

```
def train(trainfile, validationfile, params, modelname) :
    ## learns a NN model using trainfle as training data, and validationfile
    ## as validation data. Saves learnt model in a file named modelname
    # load pickle datasets (or parse if needed)
    traindata = Dataset(trainfile)
    valdata = Dataset(valfile)

    # create indexes from training data
    codes  = Codemaps(traindata, params)
    # encode datasets
    train_loader = encode_dataset(traindata, codes, params)
    val_loader = encode_dataset(valdata, codes, params)

    # build network
    network = ddiCNN(codes)

    # save indexs
    os.makedirs(modelname,exist_ok=True)
    torch.save(network, os.path.join(modelname,"network.nn"))
    codes.save(os.path.join(modelname,"codemaps"))
    # train each epoch, keep the best model on validation
    best = 0
    for epoch in range(params["epochs"]):
       train(network, epoch, train_loader)
       acc = validation(network, val_loader)
       if acc>best :
          best = acc
          torch.save(network, os.path.join(modelname,f"network.nn"))
```

Outline

Classifier - Main program

```
def predict(modelname, datafile, params, outfile) :
    '''
    Loads a NN model from file modelname and uses it to extract
    drug interactions form datafile
    '''
    # Load model
    model = torch.load(os.path.join(modelname,"network.nn"),
                       map_location=torch.device(used_device))
    model.eval()
    # load indexes
    codes = Codemaps(os.path.join(modelname,"codemaps"), params)
    # load data to classify
    testdata = Dataset(datafile)
    test_loader = encode_dataset(testdata, codes, params)

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

    # output results
    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 the whole dataset with Spacy takes some time, so it is convenient to run it once and for all:

```
$ python3 run.py parse
```

This will create pickle files for train and devel data in `preprocessed` folder, so they can be used by the feature extractor.

Auxiliary classes - Dataset


```
class Dataset:
   ## constructor:
   ## If 'filename' is a '.pck' file, load data set pickle file
   ## Otherwise, assume 'filename' is an xml file: Tokenize
   ## each sentence, and store a list of (sentence, entity pair).
   ## For each (sentence, entity pair), store token information,
   ## masking all entities in the sentence.
   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)
```

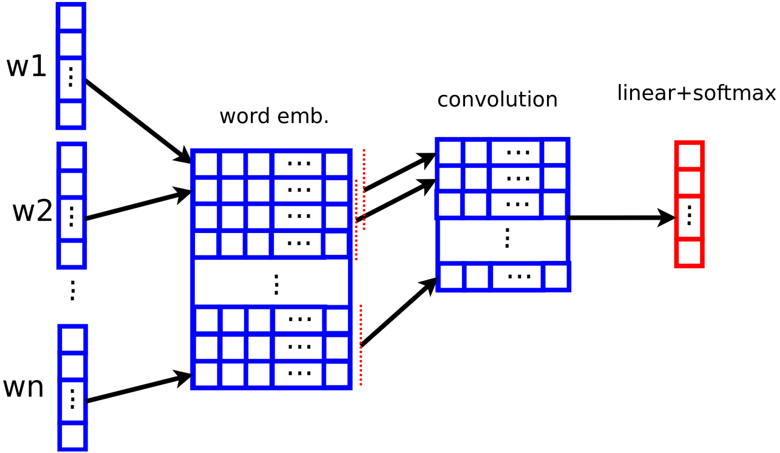
Required functions - network.py

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

    def __init__(self, codes) :
        super(ddiCNN, self).__init__()
        # get sizes from index
        n_words = codes.get_n_words()
        n_labels = codes.get_n_labels()
        max_len = codes.maxlen
        # create embedding layer
        embW_sz = 100
        self.embW = nn.Embedding(n_words, embW_sz, padding_idx=0)
        # create CNN layer
        cnn_out_sz = 32
        self.cnn = nn.Conv1d(embW_sz, cnn_out_sz, kernel_size=2, stride=1,
       padding='same')
        self.drop2 = nn.Dropout(0.2)
        # final classification layer
        self.out = nn.Linear(cnn_out_sz*self.max_len, n_labels)

    def forward(self, w):
        # run layers on given data
        x = self.embW(w)      # apply embedding layer
        x = x.permute(0,2,1)  # set shape appropriate for CNN input
        x = self.cnn(x)       # apply CNN
        x = func.relu(x)      # activation function
        x = x.flatten(start_dim=1)   # set shape appropiate for linear input
        x = self.out(x)       # final classification layer
        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.
- If you add extra input layers (e.g. lemma, pos, lowercase word, etc) 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 pairs have label "`null`" (no interaction). 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:

- Experiment with architecture variations.
- Experiment with different learning hyperparameters.
- Experiment with different input information
- Keep track of tried variants and parameter combinations.

What you should **NOT** do:

- Get insights about errors from `devel` dataset. You have `train` for that. `devel` is only used to evaluate the performance of a given configuration.
- Select architectures or hyperparameters based on system performance on `test` dataset. You have `devel` for that. `test` is only used to evaluate model generalization ability once the best configuration has been chosen.

Deliverables

At the end of the DDI task, you will need to deliver a single report on the work carried out on DDI-ML, DDI-NN, and DDI-LLM systems

So, during the development and experimentation on DDI-NN:

- keep track of tried/discarded architectures/hyperparameters
- keep track of tried/discarded input information.
- Record obtained results in the different experiments, and compile the information you'll later need to elaborate the report.