Advanced Human Language Technologies

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Outline

Task 1.2 - NERC using neural networks

Assignment

Write a python program that parses the given XML and recognizes and classifies drug names. The program must use a neural network approach.

```
$ python3 ./nn-NER.py devel.xml result.out
DDI-DrugBank.d278.s0|0-9|Enoxaparin|drug
DDI-DrugBank.d278.s0|93-108|pharmacokinetics|group
DDI-DrugBank.d278.s0|113-124|eptifibatide|drug
DDI-MedLine.d88.s0|15-30|chlordiazepoxide|drug
DDI-MedLine.d88.s0|33-43|amphetamine|drug
DDI-MedLine.d88.s0|49-55|cocaine|drug
DDI-MedLine.d88.s1|82-95|benzodiazepine|drug
...
```

Outline

General Structure

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

- B-I-O schema
- 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 proper input 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 learner will need to create and store *index* dictionaries to be able to map the index number assigned to each word, label, or any other used piece of information. See class *Codemaps*.

Outline

Outline

Learner - Main program

```python
def train(trainfile, validationfile, params, modelname) :
   '''
   learns a NN model using trainfile as training data, and validationfile
   as validation data. Saves learnt model in a file named modelname
   '''
   # load train and validation data in a suitable form
   traindata = Dataset(traindir)
   valdata = Dataset(validationdir)

   # 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 = nercLSTM(codes)

   # save indexs
   os.makedirs(modelname,exist_ok=True)
   codes.save(modelname+"/codemaps")

   # train network, keep the best performing model
   best = 0
   for epoch in range(params['epochs']):
      acc = train(network, epoch, train_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 'modelname' and uses it to extract drugs
    in datafile. Saves results to 'outfile' in the appropriate format.
    '''
    # 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 annotate
    testdata = Dataset(datafile)
    test_loader = encode_dataset(testdata, codes, params)
    # run each sentence through the NN, get results
    Y = []
    for X in test_loader:
       y = model.forward(*X)
       Y.extend([[codes.idx2label(torch.argmax(w)) for w in s] for s in y] )

    # print results
    output_entities(testdata, Y, codes, outfile)
```

Outline

Auxiliary classes - Dataset

```
class Dataset:
   ## constructor: parses datafile XML file, tokenizes each sentence, and
   ## stores a list of sentences, as well as ground truth for each token
   def __init__(self, datafile)

   ## iterator to get all sentences in the data set.
   ## for each sentence returns a triplet (text, tokens, labels)
   def sentences(self)

   ## iterator to get ids for sentence in the data set
   def sentence_ids(self)

   ##  get tokens for one  given its id
   def get_sentence_tokens(self, sid) :
   ##  get text for one sentence given its id
   def get_sentence_text(self, sid) :
   ##  get labels for one sentence given its id
   def get_sentence_labels(self, sid) :
  '''
```

Auxiliary classes - Codemaps

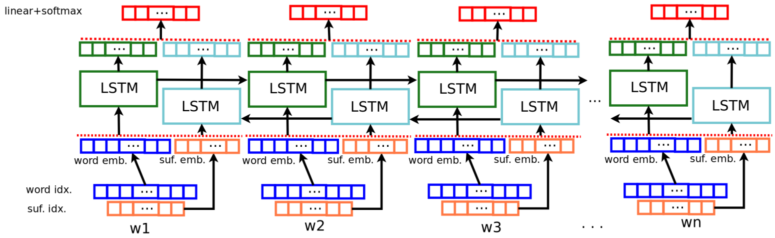

```
class Codemaps :
    # Constructor: create code mapper either from training data, or
    #              loading codemaps from given file.
    #              If 'data' is a Dataset, and lengths are not None,
    #              create maps from given data.
    #              If data is a string (file name), load maps from file.
    def __init__(self, data, params)
    # 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 nercLSTM(nn.Module):
   def __init__(self, codes):
      super(nercLSTM, self).__init__()
      # get sizes from index
      n_words = codes.get_n_words()
      n_sufs = codes.get_n_sufs()
      n_labels = codes.get_n_labels()
      # create embedding layers
      embW_sz, embS_sz = 100, 50
      self.embW = nn.Embedding(n_words, embW_sz)
      self.embS = nn.Embedding(n_sufs, embS_sz)
      self.dropW = nn.Dropout(0.1)
      self.dropS = nn.Dropout(0.1)
      # create LSTM layer + final linear classification layer
      lstm_in_sz, lstm_out_sz = embW_sz+embS_sz, 200
      self.lstm = nn.LSTM(lstm_in_sz, lstm_out_sz,
                          bidirectional=True, batch_first=True)
      self.out = nn.Linear(2*lstm_out_sz, n_labels)

   def forward(self, w, s):
      x = self.embW(w)  # apply embedding layers to input
      y = self.embS(s)
      x = self.dropW(x) # apply dropout to embeddings output
      y = self.dropS(y)
      # concatenate embeddigns for word + suffix
      x = torch.cat((x, y), dim=2)
      x = self.lstm(x)[0]  # feed concatenated vector to LSTM
      x = self.out(x)  # feed LSTM output to classification layer
      return x
```

Network architecture



Outline

Build a good NN-based drug NERC

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

- Embedding layers dimensions
- Initialitzing word embeddings with available pretrained models
- Max length and suffix length values
- Number and/or size of LSTM layers
- Used optimizer, learning rate, batch size, ...
- Number and kind of layers or activation functions
- Additional input layers (maybe with embeddings). Attention: This will require extending class `Codemaps` to handle the codes of added input layers.
  - different length suffixes and/or prefixes
  - PoS tags
  - lemmas
  - feature layer (with information about capitalization, dashes, presence in external resources, etc)

Build a good NN-based drug NERC

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 98%. However, this is due to the fact that most of the words have label "O" (non-drug). Accuracy values around 98% roughly correspond to F_1 values under 25%. To get a reasonable F_1, validation set accuracy should reach about 99.5%.

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

So, during the development and experimentation on NERC-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.