Sequence Labeling

Lab session

In this session we will build an HMM model for PoS-tagging and then CRF and neural models for Named Entity Recognition.

1. Building a simple HMM

In this first part of the lab we will build a very simple bigram HMM using MLE estimates over the Brown corpus, which is Part-of-Speech tagged.

The corpus can be imported as follows:

```
>>> from nltk.corpus import brown
```

The corpus is in the form of sequences of sentences, where each sentence is made by a sequence of pairs (word, POS-tag), like this:

```
>>> corpus = brown.tagged_sents()
```

```
>>> [[(u'The', u'AT'), (u'Fulton', u'NP-TL'), (u'County', u'NN-TL'), (u'Grand',
u'JJ-TL'), (u'Jury', u'NN-TL'), (u'said', u'VBD'), (u'Friday', u'NR'), (u'an',
u'AT'), (u'investigation', u'NN'), (u'of', u'IN'), (u"Atlanta's", u'NP$'),
(u'recent', u'JJ'), (u'primary', u'NN'), (u'election', u'NN'), (u'produced',
u'VBD'), (u'``', u'``'), (u'no', u'AT'), (u'evidence', u'NN'), (u"''", u"''"),
(u'that', u'CS'), (u'any', u'DTI'), (u'irregularities', u'NNS'), (u'took',
u'VBD'), (u'place', u'NN'), (u'.', u'.')], ...]
```

We recall (from the course) that a Hidden Markov Model is composed by:

$Q = q_1 q_2 \ldots q_N$	a set of N **states**
$A = a_{11} \ldots a_{ij} \ldots a_{NN}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{N} a_{ij} = 1 \quad \forall i$
$O = o_1 o_2 \ldots o_T$	a sequence of T **observations**, each one drawn from a vocabulary $V = v_1, v_2, \ldots, v_V$
$B = b_i(o_t)$	a sequence of **observation likelihoods**, also called **emission probabilities**, each expressing the probability of an observation o_t being generated from a state i
$\pi = \pi_1, \pi_2, \ldots, \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$

In our case, the set of states Q is made by the vocabulary of labels (the POS-tags). The vocabulary V corresponds to the word vocabulary. Observations correspond to the sentences in the corpus.

1. Extract Q and V from the corpus and use their size to initialize the probability matrices;

We will now split the corpus into a training and a test part, with the test part containing the last 10 sentences of the Brown corpus:

```
training = corpus[:-10]
testing = corpus[-10:]
```

2. Calculate pi, A and B from the training part of the corpus; since we are considering bigrams, the probabilities of the transition matrix will be calculated as MLE estimate: c(t-1, t) / c(t-1)
3. Unseen probabilities will be approximated using lidstone smoothing (k=0.1)

Once these matrices have been calculated, the model is ready. To apply the model to a sentence, we need to decode using the Viterbi algorithm. To help you, a Viterbi code has been uploaded to the moodle (*note: to use this version you need to assign a numeric id to each word and tag*):

```
"""
params is a triple (pi, A, B) where
pi = initial probability distribution over states
A = transition probability matrix
B = emission probability matrix
observations is the sequence of observations (in our case, the observed words)
the function returns the optimal sequence of states and its score
"""
def viterbi(params, observations):
    pi, A, B = params
    M = len(observations)
    S = pi.shape[0]
    alpha = np.zeros((M, S))
    alpha[:,:] = float('-inf') #cases that have not been treated
    backpointers = np.zeros((M, S), 'int')
    # base case
    alpha[0, :] = pi * B[:,observations[0]]
    # recursive case
    for t in range(1, M):
        for s2 in range(S):
            for s1 in range(S):
                score = alpha[t-1, s1] * A[s1, s2] * B[s2, observations[t]]
                if score > alpha[t, s2]:
                    alpha[t, s2] = score
                    backpointers[t, s2] = s1
    ss = []
    ss.append(np.argmax(alpha[M-1,:]))
    for i in range(M-1, 0, -1):
        ss.append(backpointers[i, ss[-1]])
    return list(reversed(ss)), np.max(alpha[M-1,:])
```

Example of use:

```
>>> predicted, score = viterbi((pi, A, B), map(lambda (w,t): w, sent))

>>> print '%20s\t%5s\t%5s' % ('TOKEN', 'TRUE', 'PRED')
>>> for (wi, ti), pi in zip(sent, predicted):
        print '%20s\t%5s\t%5s' % (word_names[wi], tag_names[ti], tag_names[pi])
```

```
               TOKEN     TRUE  PRED
                 you     PPSS  PPSS
               can't      MD*   MD*
                very       QL    QL
                well       RB    UH
               sidle       VB MD*-HL
```

4. Calculate Precision, Recall and F1-measure on the test part of the corpus
5. Transform the model into a trigram HMM and re-evaluate the results

2. Using NLTK's HMM implementation

We don't need to build HMM from scratch. NLTK includes some models for sequence labeling, including HMMs.

```
>>> from nltk.tag import hmm
trainer = HiddenMarkovModelTrainer(tag_set, symbols)
    hmm = trainer.train_supervised(
      training_seq, estimator=lambda fd, bins: LidstoneProbDist(fd, 0.1, bins),
    )
print('Testing...')
hmm.test(test_seq, verbose=True)
```

6. Use the built-in NLTK HMM model to carry out the experiments proposed in the previous section

3. NER using Conditional Random Fields

For this exercise we will need to use the sklearn_crfsuite package. If it is not installed, it can be installed using pip with `pip install sklearn-crfsuite`.

First of all, you need to download the dataset. This dataset (from Kaggle) is available on the moodle and is named *ner_dataset.csv* .

Pandas can be used to read the content of the csv:

```
>>> import pandas as pd
>>> data = pd.read_csv("ner_dataset.csv", encoding="latin1")
>>> data = data.fillna(method="ffill") #repeat sentence number on each row
```

At this point it is possible to obtain the vocabulary of words in the dataset:

```
>>> words = list(set(data["Word"].values)) #vocabulary V
>>> n_words = len(words)
```

In the file crf_helper.py there is some code that can help you to read the sentences and to extract features in the format expected by *crfsuite*. In particular, the `SentenceGetter` class transforms sentences into sequences of (word, POS, tag) triples:

```
>>> [(u'Thousands', u'NNS', u'O'), (u'of', u'IN', u'O'), (u'demonstrators',
u'NNS', u'O'), (u'have', u'VBP', u'O'), (u'marched', u'VBN', u'O'),
(u'through', u'IN', u'O'), (u'London', u'NNP', u'B-geo'), (u'to', u'TO', u'O'),
(u'protest', u'VB', u'O'), (u'the', u'DT', u'O'), (u'war', u'NN', u'O'),
(u'in', u'IN', u'O'), (u'Iraq', u'NNP', u'B-geo'), (u'and', u'CC', u'O'),
(u'demand', u'VB', u'O'), (u'the', u'DT', u'O'), (u'withdrawal', u'NN', u'O'),
(u'of', u'IN', u'O'), (u'British', u'JJ', u'B-gpe'), (u'troops', u'NNS', u'O'),
(u'from', u'IN', u'O'), (u'that', u'DT', u'O'), (u'country', u'NN', u'O'),
(u'.', u'.', u'O')]
```

The word2features function is dedicated to generate the features for the CRF, for example look at this extract:

```
if i > 0: #features related to preceding word/tag
        word1 = sent[i-1][0]
        postag1 = sent[i-1][1]
        features.update({
            '-1:word.lower()': word1.lower(), #previous word
            '-1:postag': postag1, #POS tag of previous word
        })
```

As you can see, we are using POS tags to help NER classification and we are looking at previous word (as we would do in a bigram HMM model).

We can now build the features and label vectors:

```
>>> X = [sent2features(s) for s in sentences]
>>> y = [sent2labels(s) for s in sentences]
```

An implementation of CRF is provided by the package sklearn_crfsuite:

```
>>> from sklearn_crfsuite import CRF
>>> crf = CRF(algorithm='lbfgs',
          max_iterations=100)
```

for a model with the gradient descent algorithm and a limit to 100 iterations (you can eventually reduce them if you find that the model is too slow).

Now we build the model and evaluate it on a 66/33 split between training and testing:

```
>>> from sklearn.model_selection import train_test_split
>>> from sklearn_crfsuite.metrics import flat_classification_report
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
random_state=22)
>>> crf.fit(X_train, y_train)
>>> pred=crf.predict(X_test)
>>> report = flat_classification_report(y_pred=pred, y_true=y_test)
>>> print(report)
```

The report will show accuracy stats for all classes, but we are not interested in the 'O' class. Focus on B-per, I-per, B-geo, I-geo and B-org, I-org (you can easily calculate the F-score for these classes, eventually with the help of https://pypi.org/project/seqeval/ package).

7. Propose a set of features to improve the results for PER, GEO and ORG named entities classes. Indicate the gain over the results obtained with the basic (equivalent to HMM) features.

4. NER using Deep Learning + CRFs

You can keep the code for part 3 until the sentence getter. When you have all the sentences, you will need to have dictionaries to convert words and tags to indices:

```
>>> word2idx = {w: i + 1 for i, w in enumerate(words)}
>>> tag2idx = {t: i for i, t in enumerate(tags)}
```

Now we can create the input vectors for sentences and the output vectors (labels) using the pad_sequences function from Keras. We set maxlen at 75:

```
>>> from keras.preprocessing.sequence import pad_sequences
>>> max_len=75
>>> X = [[word2idx[w[0]] for w in s] for s in sentences]
>>> X = pad_sequences(maxlen=max_len, sequences=X, padding="post",
value=n_words-1)
>>> y = [[tag2idx[w[2]] for w in s] for s in sentences]
>>> y = pad_sequences(maxlen=max_len, sequences=y, padding="post",
value=tag2idx["O"])
```

The labels must also be transformed to categorical:

```
>>> from keras.utils import to_categorical
>>> y = [to_categorical(i, num_classes=n_tags) for i in y]
```

We split train and test as in Section 3:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
random_state=22)
```

The model is built as follows:

```
from keras.models import Model, Input
from keras.layers import LSTM, Embedding, Dense, TimeDistributed, Dropout,
Bidirectional
from keras_contrib.layers import CRF
input = Input(shape=(max_len,))
model = Embedding(input_dim=n_words + 1, output_dim=20,
                  input_length=max_len, mask_zero=True)(input)  # 20-dim
embedding
model = Bidirectional(LSTM(units=50, return_sequences=True,
                           recurrent_dropout=0.1))(model)  # variational biLSTM
model = TimeDistributed(Dense(50, activation="relu"))(model)
crf = CRF(n_tags)
out = crf(model)
```

```
model = Model(input, out)
model.compile(optimizer="rmsprop", loss=crf.loss_function,
metrics=[crf.accuracy])
```

We need now to build the model and test it:

```
>>> history = model.fit(X_train, np.array(y_train), batch_size=32, epochs=5,
                    validation_split=0.33, verbose=1)
>>> test_pred = model.predict(X_test, verbose=1)
```

We can now calculate F-measure.

8. Exercise: include GloVe embeddings in the input layer to take into account semantic similarity between words

Reminder: to load GloVe embeddings

```
glove_dir = '/Users/fchollet/Downloads/glove.6B' #substitute with the right
directory for your installation
embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
print('Found %s word vectors.' % len(embeddings_index))

embedding_dim = 100 #if you use other embeddings, introduce the right size here
embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    if i < max_words:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector
```

Pre-trained vectors can be injected into a layer in this way:

```
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```