# POS_MEMM (Optional)

April 15, 2020

# 1 Introduction

### 1.0.1 Due March 17th, 23:59

In this homework you will be implementing a LSTM model for POS tagging.

You are given the following files: - `POS_NEMM.ipynb`: Notebook file for NEMM model (Optional) - `POS_LTML.ipynb`: Notebook file for MTML model - `train.txt`: Training set to train your model - `test.txt`: Test set to report your model's performance - `tags.csv`: Treebank tag universe - `sample_prediction.csv`: Sample file your prediction result should look like - `utils/`: folder containing all utility code for the series of homeworks

### 1.0.2 Deliverables (zip them all)

- pdf or html version of your final notebook
- Use the best model you trained, generate the prediction for test.txt, name the output file prediction.csv (Be careful: the best model in your training set might not be the best model for the test set).
- writeup.pdf: summarize the method you used and report their performance. If you worked on the optional task, add the discussion. Add a short essay discussing the biggest challenges you encounter during this assignment and what you have learnt.

(**You are encouraged to add the writeup doc into your notebook using markdown/html langauge, just like how this notes is prepared**)

# 2 Load data

In [1]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
import os
import sys
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from scipy import sparse
import pickle

# add utils folder to path
p = os.path.dirname(os.getcwd())
```

```python
    if p not in sys.path:
        sys.path = [p] + sys.path

from utils.hw5 import load_data, save_prediction, check_path_search
from utils.general import show_keras_model
```

```
Using TensorFlow backend.
/usr/local/lib/python3.6/site-packages/tensorflow/python/framework/dtypes.py:516:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.6/site-packages/tensorflow/python/framework/dtypes.py:517:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.6/site-packages/tensorflow/python/framework/dtypes.py:518:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.6/site-packages/tensorflow/python/framework/dtypes.py:519:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.6/site-packages/tensorflow/python/framework/dtypes.py:520:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.6/site-packages/tensorflow/python/framework/dtypes.py:525:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
/usr/local/lib/python3.6/site-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:541: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy,
it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.6/site-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:542: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy,
it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.6/site-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:543: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy,
it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.6/site-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:544: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy,
it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.6/site-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:545: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy,
it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.6/site-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:550: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy,
it will be understood as (type, (1,)) / '(1,)type'.
```

```
np_resource = np.dtype([("resource", np.ubyte, 1)])
```

tags is a dictionary that maps the Treebank tag to its numerical encoding. There are 45 tags in total, plus a special tag START (tags[-1]) to indicate the beginning of a sentence.

```
In [2]: tags = pd.read_csv('tags.csv', index_col=0).tag_encode.to_dict()

        train, train_label = load_data("train.txt")
        train, dev, train_label, dev_label = train_test_split(train, train_label)
        test, _ = load_data("test.txt")

        print("Training set: %d" % len(train))
        print("Dev set: %d" % len(dev))
        print("Testing set: %d" % len(test))
```

```
Training set: 33539
Dev set: 11180
Testing set: 9955
```

# 3 Shortest Path Search

In the class we introduced the first order markov model (HMM and MEMM). In these models, the probability distribution of the current state at t is conditioned on the previous state at t-1.

Suppose: - L is the number of tags - T is the length of the sentence - $g_t$ is the predicted tag at time t

The probability of the current stage can be discribed by a (L+1)*L matrix (taking into account of the extra START tag). The model output for a given sentence should be a T*(L+1)*L matrix (Ms) where:

$$Ms[t, i, j] = p(g_t == j | g_{t-1} == i)$$

However, we need a post processing to convert the model output into the final prediction. The idea is to find such a path $\{g_0, g_1, g_2, ..., g_T\}$ so that the path probability is maximized:

$$P(g_0, g_1, ..., g_T) = \Pi_{t=1}^{n} p(g_t | g_{t-1})$$

In practice, you may find that $\Pi_{t=1}^{n} p(g_t | g_{t-1})$ diminish to zero very quickly. For numerical stability, we maximize the logarithm of the path probability:

$$lg(P(g_0, g_1, ..., g_T)) = \sum_{t=1}^{n} lg(p(g_t | g_{t-1}))$$

## 3.1 Greedy method

Heuristically one can use greedy path search, meaning to greedily choose the step that maximize the current probability.

```
In [3]: def greedy_path_search(Ms):
            """
            greedy path search to get the label prediction path

            input: np.array(T, L+1, L)
            return: list, prediction path
            """
            path = [Ms.shape[1] - 1]
            for t, M in enumerate(Ms):
                gt = np.argmax(M[path[-1]])
                path.append(gt)
            return path[1:]
```

here we load some test data, and see what is the output of this `greedy_path_search`:

```
In [4]: with open("examples.pickle","rb") as f:
            examples = pickle.load(f)
        print("example 0: ", greedy_path_search(examples[0]['Ms']))
        print("example 1: ", greedy_path_search(examples[1]['Ms']))
```

```
example 0:  [0, 0, 0, 3, 8, 5, 6, 7, 8, 7, 0, 0, 0, 7, 3, 8, 8, 10, 11, 5, 7, 3, 8,
12, 8, 13, 4, 8, 10, 3, 14, 8, 2, 15]
example 1:  [0, 10, 3, 8, 7, 0, 0, 10, 11, 0, 0, 0, 10, 16, 2, 3, 8, 8, 10, 17, 13, 6,
14, 8, 8, 14, 7, 3, 8, 7, 0, 0, 0, 15]
```

## 3.2 Viterbi path search

There is no guarrantee that the greedy search can always find the optimal solution. You can run the test below and see that for most of the time, the greedy search leads to the wrong result.

```
In [5]: check_path_search(greedy_path_search, examples)
```

```
Success: 112 / 200
```

We introduce the viterbi algorithm. Instead of choosing the maximum probability at the current step, we use DP (dynamical programming) to memorize the best paths to all the possible tags at time `t` and the corresponding probability. This algorithm has been explained in details in class. Refer to your notes if you are in doubts.

```
In [6]: def viterbi_path_search(Ms):
            """
                :param
                Ms: np.array(T, L+1, L)
                return: list, prediction path
            """
            """
                Write your code below

                Hints:
                1. You need a matrix to store the current path probabilities
                2. You need a matrix to store the backpointers to retrieve
                the path
                3. Remember that the path probability diminish to zero fast, what
                should you do for numerical stability?
            """
            #sum  ( ( | -1))
            # [ , , ]= ( == | -1== )
            T, L = Ms.shape[0], Ms.shape[1] -1
            back = np.zeros((T, L)) # at time t, with ending at gl, the current best path
        previous point
            prob = np.zeros((T, L)) # at time t, with ending at gl, the current best path total
```

```
ln(prob)
    path = np.zeros((T)).astype(int)
    prob[0,:] = np.log(Ms[0,L,:])
    back[0,:] = L
    for t in range(1,T):
        for i in range(L):
            #pmax = prob[t-1,0] + np.log(Ms[t,0,i])
            #argpmax = 0
            #for j in range(1,L):
            #    p = prob[t-1,j] + np.log(Ms[t,j,i])
            #    if p > pmax:
            #        pmax = p
            #        argpmax = j
            ps = prob[t-1,:] + np.log(Ms[t,:-1,i])
            pmax = np.max(ps)
            argpmax = np.argmax(ps)
            prob[t,i] = pmax
            back[t,i] = argpmax
    # total time complexity: T * L * L
    last = np.argmax(prob[T-1])
    path[T-1] = last
    for t in reversed(range(T-1)):
        path[t] = back[t+1, path[t+1]]
    return path.tolist()
```

In [7]:
```
### only for testing, T = 3 case #######
def bruteforce_path_search(Ms):
    T, L = Ms.shape[0], Ms.shape[1] -1
    p = np.zeros((L,L,L))
    for i0 in range(L): # t = 0, connect L and some element i0
        p[i0,:,:] += np.log(Ms[0, L, i0])
        for i1 in range(L): # t = 1
            p[i0,i1,:] += np.log(Ms[1, i0, i1])
            for i2 in range(L): # t = 2
                p[i0,i1,i2] += np.log(Ms[2, i1, i2])
    argp = np.unravel_index(p.argmax(), p.shape)
    print([np.log(Ms[0, L ,argp[0]]),np.log(Ms[1 ,argp[0], argp[1]]),np.log(Ms[2,
argp[1] ,argp[2]])])
    return argp
```

In [8]:
```
check_path_search(viterbi_path_search, examples)
```

```
Success: 200 / 200
```

# 4 MEMM (1st order)

The training clas we use here is a little similar to the one we have in HW01. I provide the code for you so you can focus on the important part of the model, `featurization`. First go through the code and understand how it works. It helps you to accomplish the work after.

## 4.1 Base Code

In [9]:
```
from collections import defaultdict
from sklearn.linear_model import LogisticRegression
class BaseFeaturizer:
    """
    A template for the Featurizer. Later you will have to create your own featurizer, it
has to follow
    the class signature as `BaseFeaturizer`.
    In the training class, this will be used like this:
```

```
    featurize = BaseFeaturizer()
    features = featurize(t, sent, prev_tag)

    see below for the meaning of parameters
    """
    def __call__(self, t, sent, prev_tag, **kwargs):
        """
        input:
            t: int, the current index of the word in the sentence
            sent: list[string], the entire current sentence
            prev_tag: string, previous word's tag. If t=0, prev_tag would be `START`
                (remember in First order MEMM, the entire sentence both before and after
the current
                word, together with the previous 1 tage can be used to construct
features. )
            kwargs: Use this as needed to pass extra parameters
        """
        features = dict()
        features['PREV_TAG_%s' % prev_tag] = 1
        features['UNIGRAM_%s' % sent[t]] = 1

        return features


class FOMEMM:
    """
    First order maximum entropy Markov model
    """
    def __init__(self, Featurizer=BaseFeaturizer, path_search=viterbi_path_search,
                 tag_vocab=tags, **kwargs):
        """
        input:
            Featurizer: BaseFeaturizer class
            path_search: function for path search
            tag_vocab: tag dictionary, you will less likely need to change this

            kwargs: Use as needed to pass extra parameters
        """
        self.feature_vocab = {}
        self.featurize = Featurizer()
        self.path_search = path_search
        self.tag_vocab = tag_vocab
        self.reverse_tag_vocab = {v:k for k, v in tag_vocab.items()}
        self.model = None

        """
        Feel free to add code here as you need
        """

    def collect_features(self, X, y):
        """
        Create feature vocabulary from all input data
        input:
            X: list of sentences
            y: list of list of tags
        """
        feature_counts = defaultdict(int)
        for sent, labs in zip(X, y):
            for t in range(len(labs)):
                if t == 0:
                    prev_tag = "START"
                else:
                    prev_tag = labs[t-1]

                # Use next_tag
                if t == len(labs) - 1:
                    next_tag = "END"
                else:
```

```
                next_tag = labs[t+1]

            feats = self.featurize(t, sent, prev_tag, next_tag = next_tag)

            #feats = self.featurize(t, sent, prev_tag)

            for f in feats:
                feature_counts[f] += 1

        """
        Below I just use all the features I collect to construct
        the feature_vocab. Feel free to add code here as needed.
        """

        feature_vocab = {k:i+1 for i, k in enumerate(feature_counts.keys())}
        feature_vocab['_UNKNOWN_'] = 0

        return feature_vocab

    def pipeline(self, X, y):
        """
        Translate all input raw data into trainable numerical data
        input:
            X: list of sentences
            y: list of list of tags
        """
        ntot = sum([len(sent) for sent in X])
        X_new = sparse.dok_matrix((ntot, len(self.feature_vocab)))

        i = 0
        for sent, labs in zip(X, y):
            for t in range(len(labs)):
                if t == 0:
                    prev_tag = "START"
                else:
                    prev_tag = labs[t-1]

                # Use next_tag
                if t == len(labs) - 1:
                    next_tag = "END"
                else:
                    next_tag = labs[t+1]

                #feats = self.featurize(t, sent, prev_tag)

                feats = self.featurize(t, sent, prev_tag, next_tag = next_tag)

                for f, v in feats.items():
                    X_new[i, self.feature_vocab[f]] = v

                i += 1

        y_new = np.array([self.tag_vocab[t] for labs in y for t in labs])

        return X_new, y_new

    def fit(self, X, y):
        """
        input:
            X: list of sentences
            y: list of list of tags
        """
        self.feature_vocab = self.collect_features(X, y)
        self.X, self.y = self.pipeline(X, y)

        self.model = LogisticRegression(C=1.0, multi_class='auto')
        self.model.fit(self.X, self.y)
```

```
        return self

    def predict(self, X):
        results = []
        L = len(self.tag_vocab) - 1
        for j, sent in enumerate(X):
            T = len(sent)
            Y_pred = np.zeros((T, L+1, L))
            for t in range(len(sent)):
                # We need to make one prediction for each possible prev_tag
                for prev_tag, i in self.tag_vocab.items():
                    feats = self.featurize(t, sent, prev_tag)
                    X0 = sparse.lil_matrix((1, len(self.feature_vocab)))
                    for f, v in feats.items():
                        X0[0, self.feature_vocab.get(f, 0)] = v

                    prob = self.model.predict_proba(X0)
                    Y_pred[t, i, :-1] = prob

            tag_path = self.path_search(Y_pred)
            results.append([self.reverse_tag_vocab[t] for t in tag_path])

            print("Generation predictions, %.3f%% complete." % (j*100./len(X)),
end="\r")

        return results

    def score(self, X, y):
        tot_tokens = sum([len(sent) for sent in X])
        tot_sents = len(X)

        y_pred = self.predict(X)

        right_tokens, right_sents = 0, 0

        for labs0, labs1 in zip(y, y_pred):
            right_sents += (labs0 == labs1)
            for lab0, lab1 in zip(labs0, labs1):
                right_tokens += (lab0 == lab1)

        return {"tokens" : 1.0 * right_tokens / tot_tokens,
                "sentences": 1.0 * right_sents / tot_sents}
```

You can check how the model works with the base featurizer:

```
In [10]: fomemm = FOMEMM().fit(train, train_label)
         fomemm.score(dev, dev_label)
```

```
/usr/local/lib/python3.6/site-packages/sklearn/linear_model/_logistic.py:940:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:23: RuntimeWarning:
divide by zero encountered in log
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:34: RuntimeWarning:
divide by zero encountered in log


Generation predictions, 99.991% complete.
```

[10]: {'tokens': 0.9147770798222767, 'sentences': 0.3206618962432916}

## 4.2 Implement your own featurizer:

```
In [11]: class BetterFeaturizer(BaseFeaturizer):
             """
             A template for the Featurizer. Later you will have to create your own featurizer, it
         has to follow
             the class signature as `BaseFeaturizer`.
             In the training class, this will be used like this:

             featurize = BaseFeaturizer()
             features = featurize(t, sent, prev_tag)

             see below for the meaning of parameters
             """
             def __call__(self, t, sent, prev_tag, **kwargs):
                 """
                 input:
                     t: int, the current index of the word in the sentence
                     sent: list[string], the entire current sentence
                     prev_tag: string, previous word's tag. If t=0, prev_tag would be `START`
                         (remember in First order MEMM, the entire sentence both before and after
         the current
                         word, together with the previous 1 tage can be used to construct
         features. )
                     kwargs: Use this as needed to pass extra parameters
                 """
                 """
                 Implement your code here
                 """
                 features = dict()
                 features['PREV_TAG_%s' % prev_tag] = 1
                 features['UNIGRAM_%s' % sent[t]] = 1
                 features['SENT_LEN_%d' % len(sent)] = 1
                 for key, value in kwargs.items():
                     features['%s' % value] = 1
                 return features
```

```
In [12]: fomemm = FOMEMM(Featurizer=BetterFeaturizer).fit(train, train_label)
         fomemm.score(dev, dev_label)
```

```
/usr/local/lib/python3.6/site-packages/sklearn/linear_model/_logistic.py:940:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:23: RuntimeWarning:
divide by zero encountered in log
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:34: RuntimeWarning:
divide by zero encountered in log


Generation predictions, 99.991% complete.
```

[12]: {'tokens': 0.9082513022828251, 'sentences': 0.29704830053667264}

### 4.3 Save your model prediction

```
In [13]: prediction = fomemm.predict(test)
         save_prediction(prediction,"MEMM_prediction.csv")
```

```
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:23: RuntimeWarning:
divide by zero encountered in log
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:34: RuntimeWarning:
divide by zero encountered in log


Generation predictions, 99.990% complete.
```

```
In [ ]:
```