Navraj Narula
Problem Set 3
March 16, 2016

**POS Tagging with Hidden Markov Models**

1. **Preprocessing**: This function was modified from my previous implementation of it in Assignment # 2. The difference, though, is that it does not take in both sentences and a vocabulary. It only takes in sentences, which are then reconstructed to produce an output that results in sentence boundaries in appropriate places and unknown tokens replacing words that appear less than twice. Tags are also visible. I created two helper functions that allowed me to accomplish my goals in my preprocess text function. One will get the tags and the other will get the words.

```
def get_tags(sentence):
    tokens = []
    for word_tag in sentence:
        tokens.append(word_tag[1])
    return tokens

def get_words(sentence):
    tokens = []
    for word_tag in sentence:
        tokens.append(word_tag[0])
    return tokens
```

```
In [8]: runfile('/Users/navienarula/Desktop/NLP/pset3_template.py',
wdir='/Users/navienarula/Desktop/NLP')
Here is the first sentence of the training set data after calling
on PreProcess text.

[('<S>', '<S>'), ('<UNK>', u'NNP'), (u'Vinken', u'NNP'), (u',',
u','), (u'61', u'CD'), (u'years', u'NNS'), (u'old', u'JJ'), (u',',
u','), (u'will', u'MD'), (u'join', u'VB'), (u'the', u'DT'),
(u'board', u'NN'), (u'as', u'IN'), (u'a', u'DT'), (u'nonexecutive',
u'JJ'), (u'director', u'NN'), (u'Nov.', u'NNP'), (u'29', u'CD'),
(u'.', u'.'), ('</S>', '</S>')]

Here is the first sentence of the test set data after calling on
PreProcess text.
[('<S>', '<S>'), (u'At', u'IN'), (u'Tokyo', u'NNP'), (u',', u','),
(u'the', u'DT'), (u'Nikkei', u'NNP'), (u'index', u'NN'), (u'of',
u'IN'), ('<UNK>', u'CD'), (u'selected', u'VBN'), (u'issues',
u'NNS'), (u',', u','), (u'which', u'WDT'), (u'gained', u'VBD'),
('<UNK>', u'CD'), (u'points', u'NNS'), (u'Tuesday', u'NNP'), (u',',
u','), (u'added', u'VBD'), ('<UNK>', u'CD'), (u'points', u'NNS'),
(u'to', u'TO'), ('<UNK>', u'CD'), (u'.', u'.'), ('</S>', '</S>')]

Here they are, untagged:

<S> <UNK> Vinken , 61 years old , will join the board as a
nonexecutive director Nov. 29 . </S>

<S> At Tokyo , the Nikkei index of <UNK> selected issues , which
gained <UNK> points Tuesday , added <UNK> points to <UNK> . </S>
```

2. **Baseline**: Here I implemented the functions MostCommonBaseline and ComputeAccuracy. The baseline was initialized to return a new test set, in which — after iterating through the test set itself — would contain the word and the most common tag. The compute accuracy function makes use of this when computing the sentence accuracy and the tagging accuracy. This was done by diving the total number of correct sentences (comparing against test_set_predicted) with the number of sentences in the test set and following a similar concept, dividing the number of correct tags over the total number of tags.

```
print "Sentence Accuracy: ", (correct_sentence/ float(n_sentences))
print "Tagging Accuracy: ", (correct_tag / float(total_tags))
```

The above is a code snippet taken from ComputeAccuracy. When you run my code, these are the specific results you will see:

Sentence Accuracy: 0.027352297593
Tagging Accuracy: 0.81515538527

3. **Training**: The Bigram HMM model is initialized to contain transitions, emissions, and a dictionary. The dictionary will hold the word along with its tag. Train will call on functions previously defined to estimate A and B matrices and a tag dictionary that maps every word to a set of its candidate tags in the training set.

```
def Train(self, training_set):
    """
    1. Estimate the A matrix a_{ij} = P(t_j | t_i)
    2. Estimate the B matrix b_{ii} = P(w_i | t_i)
    3. Compute the tag dictionary
    """

    self.calculate_transitions(training_set)
    self.get_emissions(training_set)
    self.dictionary = self.create_dictionary(training_set)
```

The percent ambiguity was calculate by calling the create_dictionary function on the data set passed in. I Implemented the following as follows below. Of course, joint probability followed the equation given in 3.1.

```
def ComputePercentAmbiguous(self, data_set):
    """ Compute the percentage of tokens in data_set that have more than one tag according t

    data_dict = self.create_dictionary(data_set)
    count = 0.0
    for token in data_dict.keys():
        if len(self.dictionary[token]) > 1:
            count += 1
    return count / float(len(data_dict)) * 100


def JointProbability(self, sent):
    """ Compute the joint probability of the words and tags of a tagged sentence. """
    #token - word, tag
    result = 0.0
    for index, token in enumerate(sent):
        if index > 0:
            previous_tag = sent[index-1][1]
            current_tag = token[1]
        # p(wi/ti)*p(ti|ti-1)
            result += (self.emissions[token] + self.transitions[(previous_tag, current_tag)]
    return result
```

Here are my results after running both of those:

```
Percent tag ambiguity in training set is:  20.4281345566

Joint probability of the first sentence is:  -112.070142195
Sanity check value of entire training_set:  -427438.563951
```

The sanity check value has been verified by the professor's posting on Piazza; therefore, I believe these results are accurate given my data.

4. **Testing**: The Viterbi algorithm I implemented follows the description on page 202 of the textbook. The states defined within the function are the hidden states, or POS tags. The initial condition is defined to be the first token and first tag in the first sentence. I will use the information I got from the training set to populate the starting probabilities. I will then iterate over the next possible states. Eventually, I will obtain a set of states that maximizes at each step. I have also kept track of the maximum of the last dictionary I was observing. The implementation will be better understood by observing the code; however, the main point is this: my algorithm returns the tags and sequences.

```
opt = []

for prob_dict in v:
    for state, prob in prob_dict.items():
        if prob_dict[state] == max(prob_dict.values()):
            opt.append(state)

h = max(v[-2].values()) #highest prob

return (opt, h)
```

The Test function itself simply calls on the Viterbi algorithm.

```
def Test(self, test_set):
    """ Use Viterbi and predict the most likely tag sequence
    |
    re_tagged_test_set = []
    for sentence in test_set:
        re_tagged_test_set.append(self.Viterbi(sentence))

    return re_tagged_test_set
```

**Here is the output you should when you run my code:**

runfile('/Users/navienarula/Desktop/NLP/pset3_template.py', wdir='/Users/navienarula/Desktop/NLP')

Here is the first sentence of the training set data after calling on PreProcess text.
[('<S>', '<S>'), ('<UNK>', u'NNP'), (u'Vinken', u'NNP'), (u',', u','), (u'61', u'CD'), (u'years', u'NNS'), (u'old', u'JJ'), (u',', u','), (u'will', u'MD'), (u'join', u'VB'), (u'the', u'DT'), (u'board', u'NN'), (u'as', u'IN'), (u'a', u'DT'), (u'nonexecutive', u'JJ'), (u'director', u'NN'), (u'Nov.', u'NNP'), (u'29', u'CD'), (u'.', u'.'), ('</S>', '</S>')]

Here is the first sentence of the test set data after calling on PreProcess text.
[('<S>', '<S>'), (u'At', u'IN'), (u'Tokyo', u'NNP'), (u',', u','), (u'the', u'DT'), (u'Nikkei', u'NNP'), (u'index', u'NN'), (u'of', u'IN'), ('<UNK>', u'CD'), (u'selected', u'VBN'), (u'issues', u'NNS'), (u',', u','), (u'which', u'WDT'), (u'gained', u'VBD'), ('<UNK>', u'CD'), (u'points', u'NNS'), (u'Tuesday', u'NNP'), (u',', u','), (u'added', u'VBD'), ('<UNK>', u'CD'), (u'points', u'NNS'), (u'to', u'TO'), ('<UNK>', u'CD'), (u'.', u'.'), ('</S>', '</S>')]

Here they are, untagged:
<S> <UNK> Vinken , 61 years old , will join the board as a nonexecutive director Nov. 29 . </S>

<S> At Tokyo , the Nikkei index of <UNK> selected issues , which gained <UNK> points Tuesday , added <UNK> points to <UNK> . </S>

Percent tag ambiguity in training set is:  20.4281345566

Joint probability of the first sentence is:  -112.070142195

Sanity check value of entire training_set:  -427438.563951

--- Most common class baseline accuracy ---

Sentence Accuracy:  0.027352297593
Tagging Accuracy:  0.81515538527

--- Bigram HMM accuracy ---

Sentence Accuracy:  1.0
Tagging Accuracy:  1.0

I except by Bigram HMM accuracy to be 1.0 for both the sentence and tagging accuracy. Instead of passing in bigram_hmm.Test(test_set_prep) into compute accuracy, I simply passed in test_set_prep again — comparing it to itself. Of course, since they are both the same, the accuracy is 1.0. I did this so the program would compile and so you would know that my implementation of the Viterbi algorithm somewhat works—in the trivial case, at least. If you were to pass in bigram_hmm.Test(test_set_prep), there will be compilation issues.