CSCI544 - Homework Assignment No 2

1. Vocabulary Creation

Created a function **VocabularyCreation** which is used to create Vocab.txt file with input as raw train data file. The function first reads the input data and creates a dictionary of unique words with the help of function **readTrainDataFile.** The created word dictionary is processed through **createVocab** which compares the count of word with the THRESHOLD value and replace rare words whose occurrences are less than a threshold (e.g. 3) with a special token '< unk >'

```
for key, value in vocab_data.items():
    if (value > THERSHOLD-1):
        Vocab[key] = value
    else:
        Vocab['< unk >'] += value
```

Function **outputVocabFile** is used to create the **Vocab.txt** file with the help of word dictionary **VOCAB**

What is the selected threshold for unknown words replacement? **Answer - 2** What are the total occurrences of special token '< unk >' after replacement? **Answer - 20011** What is the total size of your vocabulary? **Answer - 23183**

2. Model Learning

Model Learning is the process to learn an HMM from the training data. The function **ModelLearning** is used for the process of Model Learning which creates the **TRANSITION** and **EMISSION** parameter dictionary which is the probability of the following data.

$$t(s'|s) = count(s \to s') / count(s)$$
$$e(x|s) = count(s \to x) / count(s)$$

where $t(\cdot|\cdot)$ is the transition parameter and $e(\cdot|\cdot)$ is the emission parameter.

The function first inputs the train_data and data is prepared using function **data_preparation** which reads data line and convert line in format **[(word, tag)]**. For each line in data **[('', '< start >')]** is inserted at the beginning and **word, tag, _previous_tag** are extracted to calculate the required probability using the following lines of code. First we checked if the word is present in the **VOCAB** or not and if the word is not present we replace it with **'< unk >'**

```
if word not in VOCAB:
    word = '< unk >'

tag_data[tag] += 1
transition_key = (previous_tag, tag)
transition_data[transition_key] += 1

emission_key = (tag, word)
emission_data[emission_key] += 1
```

```
for transition_key in transition_data.keys():
  _tag = transition_key[0]
  TRANSITION[transition_key] = transition_data[transition_key] / tag_data[_tag]

for emission_key in emission_data.keys():
  _tag = emission_key[0]
  EMISSION[emission_key] = emission_data[emission_key] / tag_data[_tag]
```

The function creates two dictionaries for the emission and transition parameters **TRANSITION** and **EMISSION**, respectively. The first dictionary, named **TRANSITION**, contains items with pairs of (s, s′) as key and t(s′|s) as value. The second dictionary, named **EMISSION**, contains items with pairs of (s, x) as key and e(x|s) as value.

```
for key in tag_data.keys():
       if key != '< start >':
           _TAG.add(key)
```

The set of all tags except the **< start >** tag is created in **_TAG** set

How many transition and emission parameters are in your HMM?
Answer -   Transition Parameters 1392
Emission Parameters 30303

For Task 3 and Task 4 **class HiddenMarkovModel** is developed. The hmm object of the class is created with parameters dev_data and test_data.

```
hmm = HiddenMarkovModel(dev_data, test_data)
```

3. Greedy Decoding with HMM

HiddenMarkovModel class contains **GreedyDecodingHMM** function which is used to implement the greedy decoding algorithm and evaluate it on the development data and predicting the part-of-speech tags of the sentences in the test data and output the predictions in a file named **greedy.out**

The function first inputs the train_data and data is prepared using function **data_preparation** which reads data line and convert line in format **[(word, tag)]**. For each line in data **word, tag** are extracted and we checked if the word is present in the **VOCAB** or not and if the word is not present we replace it with **'< unk >'**

$$S* = \arg\max_{s_1,\ldots,s_m} p(x_1, \ldots, x_m, s_1, \ldots, s_m) = t(s_1)\prod_{j=2}^{m} t(s_j \mid s_{j-1})\prod_{j=1}^{m} e(x_j \mid s_j)$$

The above formula is used to implement the greedy decoding where the tag for the first word is predicted using the code

```
_predicted_tag = max([_tag_ for _tag_ in _TAG], key=lambda _tag_:
   TRANSITION[('< start >', _tag_)] * EMISSION[(_tag_, word)])
```

For the consecutive words in the sentence same process is repeated but previous predicted tag **_predicted_tag** is also considered

```
predicted_tag = max([__tag for __tag in _TAG], key=lambda __tag:
 TRANSITION[(_predicted_tag, __tag)] * EMISSION[(__tag, word)])
_predicted_tag = predicted_tag
```

The function then calls the **self.outputPredictionFile('greedy')** with greedy as algorithm parameters. The function processes the test_data using function **data_preparation_test** and converts the input test data in the format **[(word)]**. The prediction code for the greedy is executed by passing the each line of the processed test data to function **self.predictGreedyDecodingHMM(sentence)** which returns the line in the format **[(word, predicted_tag)]** which is used to generate the greedy.out file using the code

```
if algorithm == 'greedy':
   for sentence in _test_data:
       predictedline = self.predictGreedyDecodingHMM(sentence)
       for word_index in range(len(predictedline)):
           _word, _tag = predictedline[word_index]
           insertline = str(word_index+1) + '\t' + _word + '\t' + _tag + '\n'
           vocab.append(insertline)
       vocab.append('\n')

    with open('greedy.out', 'wt') as file:
         file.write(''.join(vocab))
```

What is the accuracy of the dev data?

Answer - Greedy Decoding with HMM: 0.9318895372130801

4. Viterbi Decoding with HMM

HiddenMarkovModel class contains **ViterbiDecodingHMM** function which is used to implement the Viterbi decoding algorithm and evaluate it on the development data and predicting the part-of-speech tags of the sentences in the test data and output the predictions in a file named **viterbi.out**

The function first inputs the train_data and data is prepared using function **data_preparation** which reads data line and convert line in format **[(word, tag)]**. For each line in data **word, tag** are extracted and we checked if the word is present in the **VOCAB** or not and if the word is not present we replace it with **'< unk >'**

$\pi[1, s] = t(s)e(x_1|s)$, and for $j > 1$,

$$\pi[j, s] = \max_{s_1 \ldots s_{j-1}} \left[\underbrace{t(s_1)e(x_1|s_1) \left(\prod_{k=2}^{j-1} t(s_k|s_{k-1})e(x_k|s_k) \right)}_{\text{The value for position } j\text{-}1} \underbrace{t(s|s_{j-1})e(x_j|s)}_{\text{Word } j} \right]$$

Initialization: for $s = 1 \ldots k$

$$\pi[1, s] = t(s)e(x_1|s)$$

The above formula is used to implement the viterbi decoding where the tag for the first word is predicted using the code. Initially **ViterbiMatrix** and **backtrack** matrix are initialize with the size **(length of sentence) x (number of tag)**

```
ViterbiMatrix = np.zeros((line_num, _tag_num), dtype=np.float64)
backtrack = np.zeros((line_num, _tag_num), dtype=int)

for tag in range(_tag_num):
        word = sentence[0][0]
        if word not in VOCAB:
              word = '< unk >'
        ViterbiMatrix[0][tag] = TRANSITION[('< start >' _tags[tag])] *
EMISSION[(_tags[tag], word)]
        backtrack[0][tag] = tag
```

For $j = 2 \ldots m$, $s = 1 \ldots k$:

$$\pi[j, s] = \max_{s' \in \{1 \ldots k\}} [\pi[j-1, s'] \times t(s|s') \times e(x_j|s)]$$

For the consecutive words in the sentence same process is repeated but previous predicted tag **_predicted_tag** is also considered

```
for word_num in range(1, line_num):
    word, tag = sentence[word_num]
    _tag_actual.append(tag)

    if word not in VOCAB:
       word = '< unk >'
    for tag in range(_tag_num):
        ViterbiMatrix[word_num][tag], backtrack[word_num][tag] =
max((ViterbiMatrix[word_num-1][prev_tag] *
     TRANSITION[(_tags[prev_tag], _tags[tag])] *
     EMISSION[(_tags[tag], word)], prev_tag) for prev_tag in range(_tag_num))
```

$$\max_{s_1 \ldots s_m} p(x_1 \ldots x_m, s_1 \ldots s_m; \underline{\theta}) = \max_{s} \pi[m, s]$$

Finally the predicted tag list is inserted into the **_tag_index_predicted** by backtracking through the matrix

```
_tag_index_predicted = [np.argmax(ViterbiMatrix[line_num-1])]
for word_num in range(line_num-2, -1, -1):
  _tag_index_predicted.insert(0,backtrack[word_num+1][_tag_index_predicted[0]])
```

The function then calls the **self.outputPredictionFile('viterbi')** with viterbi as algorithm parameters. The function processes the test_data using function **data_preparation_test** and converts the input test data in the format **[(word)].** The prediction code for the greedy is executed by passing the each line of the processed test data to function **self.predictViterbiDecodingHMM(sentence)** which returns the line in the format **[(word, predicted_tag)]** which is used to generate the viterbi.out file using the code

What is the accuracy on the dev data? **Answer - Viterbi Decoding with HMM: 0.9476816115247703**