CSCI 544 - Applied Natural Language Processing

CSCI 544 - Assignment 2

Name: Sri Manvith Vaddeboyina

USC ID: 1231409457

Importing necessary libraries/packages

In [1]:

```
import json
import numpy as np
import pandas as pd
```

Task 1: Vocabulary Creation

Reading the train data

In [2]:

```
df_train = pd.read_csv('data/train', sep='\t', names = ['index','word type','pos tag'], header = None)
```

Reading the dev data

In [3]:

```
df_dev = pd.read_csv('data/dev',sep='\t',names = ['index','word type','pos tag'], header = None)
```

Reading the test data

In [4]:

```
df_test = pd.read_csv('data/test',sep='\t',names = ['index','word type'], header = None)
```

This function - word_dic counts the number of occurences of each word in the dataframe

In [5]:

```
def word_dic(words):
    word_dict={}
    for word in words:
        if word in word_dict:
            word_dict[word] += 1
        else:
            word_dict[word] = 1
    return word_dict
```

What is the selected threshold for unknown words replacement?

I have chosen the value of threshold as 2.
I have experimented with various values of threshold ranging from 1 to 8 and found that the value of 2 giving good results on number of tags, greedy and viterbi decoding.

What is the selected threshold for unknown words replacement?

In [6]:

```
threshold_value = 2
```

In [7]:

```
print(threshold_value)
```

```
2
```

I have replaced the words with frequency less than threshold value with < *unk* > tag

In [8]:

```
words = list(df_train['word type'])
word_dict = word_dic(words)
final_dict = word_dict

final_dict['<unk>']=0
for word in word_dict.copy():
    if(final_dict[word] < threshold_value):
        final_dict['<unk>'] += final_dict[word]
        final_dict.pop(word)
```

Creating a vocabulary using the training data in the file train and output the vocabulary into a txt file named vocab.txt.

In [9]:

```
with open('vocab.txt','w') as f:
    k=final_dict['<unk>']
    line1="<unk>"+"\t"+str(0)+"\t"+str(k)
    f.write(line1)
    f.write("\n")
    word_dict=dict(sorted(final_dict.items(), key=lambda x: x[1], reverse=True))
    i=1
    for key,value in final_dict.items():
        if key!='<unk>':
            line=str(key)+"\t"+str(i)+"\t"+str(value)
            f.write(line)
            f.write("\n")
            i+=1
```

What is the total size of your vocabulary?

Total size of vocabulary - 23183

In [10]:

```
print(len(final_dict))
```

```
23183
```

What is the total occurrences of the special token < unk > after replacement?

Total occurrences of the special token < unk > after replacement - 20011

In [11]:

```
print(final_dict['<unk>'])
```

```
20011
```

Task 2: Model Learning

This function counts the occurences of each pos tag in the train data

In [12]:

```
count_tag = word_dic(df_train['pos tag'])
count_tag1 = list(word_dic(df_train['pos tag']).keys())
```

In [13]:

```
train_new=df_train[df_train['index']==1]['pos tag']
df_train_new=pd.DataFrame(train_new)
pos_counts=dict(df_train_new['pos tag'].value_counts())
```

This function calculates the first occurence probability of each pos tag

In [14]:

```
occurence_1 = {}
for k,v in pos_counts.items():
    occurence_1[k] = v/df_train_new.shape[0]
```

The functions-sent_tags_train_dev,sent_tags_test return the sentences and tags with the replaced < unk > tags

In [15]:

```
def sent_tags_train_dev(data):
    sentence_list = []
    sentence_tags = []
    tags = []
    sentence = None
    for i in list(data.values):
        if i[0] == 1:
            if sentence:
                sentence_tags.append(tags)
                sentence_list.append(sentence)
            sentence = []
            tags = []
        if i[1] not in final_dict:
            sentence.append('<unk>')
        else:
            sentence.append(i[1])
        tags.append(i[2])
    sentence_tags.append(tags)
    sentence_list.append(sentence)

    return sentence_list, sentence_tags
```

In [16]:

```
def sent_tags_test(data):
    sentence_list = []
    sentence = None
    for i in list(data.values):
        if i[0] == 1:
            if sentence:
                sentence_list.append(sentence)
            sentence = []
        if i[1] not in final_dict:
            sentence.append('<unk>')
        else:
            sentence.append(i[1])
    sentence_list.append(sentence)

    return sentence_list
```

In [17]:

```
train_sentences_list,train_tags_list = sent_tags_train_dev(df_train)
dev_sentences_list,dev_tags_list = sent_tags_train_dev(df_dev)
test_sentences_list = sent_tags_test(df_test)
```

This function takes bigrams into consideration and calculates the transmission probability

In [18]:

```
def find_transition_prob(tags_):
    bg, transition = {}, {}

    for sentence in tags_:
        for s in range(len(sentence)-1):
            key = (sentence[s], sentence[s+1])
            if key not in bg:
                bg[key] = 0
            bg[key]+=1

    for key in bg:
        length_ = count_tag[key[0]]
        value_ = bg[key]/length_
        transition[key] = value_
    return transition
```

In [19]:

```
transition_probabilities = find_transition_prob(train_tags_list)
```

This function calculates the emission probability

In [20]:

```
def find_emission_prob(train_sentences_list,train_tags_list):
    tag_word, emission = {}, {}

    for i in range(len(train_sentences_list)):
        for j in range(len(train_sentences_list[i])):
            key_ = (train_tags_list[i][j], train_sentences_list[i][j])
            if key_ not in tag_word:
                tag_word[key_] = 0
            tag_word[key_]+=1

    for key in tag_word:
        length_ = count_tag[key[0]]
        value_ = tag_word[key]/length_
        emission[key] = value_
    return emission
```

In [21]:

```
emission_probabilities = find_emission_prob(train_sentences_list,train_tags_list)
```

This function generates the hmm.json file with

Keys - transition and emission

Values - respective transition and emission probabilities

In [22]:

```
def generate_hmm_file(transition_probabilities,emission_probabilities):
    hmm_probabilities={}
    hmm_probabilities['transition_probabilities']={}
    hmm_probabilities['emission_probabilities']={}

    for key, value in transition_probabilities.items():
        hmm_probabilities['transition_probabilities'][str(key)]=value

    for key, value in emission_probabilities.items():
        hmm_probabilities['emission_probabilities'][str(key)]=value

    with open('hmm.json','w') as f:
        json.dump(hmm_probabilities,f)
```

In [23]:

```
generate_hmm_file(transition_probabilities,emission_probabilities)
```

How many transition and emission parameters in your HMM?

Transition parameters in HMM - 1351

In [24]:

```
print(len(transition_probabilities))
```

```
1351
```

Emission parameters in HMM - 30303

In [25]:

```
print(len(emission_probabilities))
```

```
30303
```

Task 3: Greedy Decoding with HMM

I have implemented Greedy Decoding algorithm with HMM below
Created a dictionary that stores keys as each word in the vocabulary and values as a dictionary of values of each (tag,word) pair where word is same the key. i.e. Each word that has different pos tag is grouped under one key.

In [26]:

```
def words_from_emission_probs():
    key_words_emission_probs = {}

    for key in emission_probabilities:
        if key[1] not in key_words_emission_probs:
            key_words_emission_probs[key[1]] = {}
        key_words_emission_probs[key[1]][key]= emission_probabilities[key]
    return key_words_emission_probs
```

In [27]:

```
key_words_= words_from_emission_probs()
```

This function returns the corresponding tag and maximum emission probability value for each word in a sentence that is looped in the greedy decoding algorithm

In [28]:

```
def find_emission_probability(word):
    tag_lst=[]
    max_emm_prob = 0
    for w in key_words_[word]:
        e_prob = key_words_[word][w]*occurence_1.get(w[0],0.0001)
        if e_prob > max_emm_prob:
            e_tag_ = w
            max_emm_prob = e_prob

    return e_tag_,max_emm_prob
```

This function returns the corresponding tag and maximum transition probability value for each word in a sentence that is looped in the greedy decoding algorithm

In [29]:

```
def find_transition_probability(word1,word2):
    tag_lst=[]
    max_trans_prob=0
    for w in key_words_[word1]:
        t_prob = key_words_[word1][w]*transition_probabilities.get((word2,w[0]),1e-4)
        if t_prob > max_trans_prob:
            t_tag_ = w
            max_trans_prob = t_prob
    return t_tag_,max_trans_prob
```

This is the main Greedy Decoding algorithm with HMM.

For each word in a sentence the algorithm tries to assign POS tag to a word that has the highest probability derived from the above two functions.

In [30]:

```
def greedy_decoding_algorithm(sentence):
    max_keys = []
    keys = []
    for ind,word in enumerate(sentence):
        max_prob = 0
        max_key = None
        if ind == 0:
            e_tag, emmision_proba = find_emission_probability(word)
            max_prob = emmision_proba
            max_key = e_tag
        else:
            prev_emission = max_keys[-1][0]
            t_tag, transition_proba = find_transition_probability(word,prev_emission)
            max_prob = transition_proba
            max_key = t_tag

        max_keys.append(max_key)

    for k in max_keys:
        keys.append(k[0])

    return keys
```

This function calculates the greedy decoding algorithm accuracy on dev data

In [31]:

```
def greedy_decoding_accuracy(sentences):
    true_cnt = 0
    total_cnt = 0

    for i,sentence in enumerate(sentences):
        greedy_pred = np.array(greedy_decoding_algorithm(sentence))
        true_pred = dev_tags_list[i]
        true_cnt += sum(greedy_pred == true_pred)
        total_cnt += len(dev_tags_list[i])

    greedy_decode_accuracy = true_cnt/total_cnt
    greedy_decode_accuracy = greedy_decode_accuracy * 100
    return greedy_decode_accuracy
```

In [32]:

```
print(greedy_decoding_accuracy(dev_sentences_list))
```

```
93.53029567117966
```

Now we find the predictions of part-of-speech tags using greedy decode algorithm on the test data

In [33]:

```
greedy_tags_pred = []
for i,sentence in enumerate(test_sentences_list):
    greedy_tags_pred += greedy_decoding_algorithm(sentence)
df_test['greedy_tags'] = greedy_tags_pred
```

We now write the output predictions in a file named greedy.out, in the same format of training data.

In [34]:

```
with open('greedy.out','w') as f:
    for i,j in enumerate(df_test.values):
        line = str(j[0])+"\t"+str(j[1])+"\t"+str(j[2])
        if j[0]==1 and i!=0:
            f.write("\n")
        f.write(line)
        f.write('\n')
```

Task 4: Viterbi Decoding with HMM

This is the main Viterbi Decoding algorithm with HMM.

The Viterbi algorithm is a dynamic programming algorithm for obtaining the maximum a posteriori probability estimate of the most likely sequence of hidden states—called the Viterbi path—that results in a sequence of observed events, especially in the context of Markov information sources and hidden Markov models (HMM).

I have used the code from wikipedia as a reference for this code

In [35]:

```
def viterbi_decoding_algorithm(each_sent):
    optimal_tags = []

    viterbi_dp = [{}]

    for each_tag in count_tag1:
        if each_sent[0] != '<unk>':
            viterbi_dp[0][each_tag] = {"prob": occurence_1.get(each_tag,1e-8) * emission_probabilities.get((each_tag,each_sent[
                                        "prev": None }

        else:
            viterbi_dp[0][each_tag] = {"prob": occurence_1.get(each_tag,1e-8),
                                        "prev": None }

    for ind in range(1, len(each_sent)):
        viterbi_dp.append({})

        for each_tag in count_tag1:
            if each_sent[ind] != '<unk>':
                max_tr_prob = viterbi_dp[ind - 1] [count_tag1[0]] ["prob"] * transition_probabilities.get((count_tag1[0],each_t

            else:
                max_tr_prob = viterbi_dp[ind - 1] [count_tag1[0]] ["prob"] * transition_probabilities.get((count_tag1[0],each_t

            prev_st_selected = count_tag1[0]

            for prev_st in count_tag1[1:]:
                tr_prob = viterbi_dp[ind - 1] [prev_st] ["prob"] * transition_probabilities.get((prev_st,each_tag),1e-8)  * em:

                if tr_prob > max_tr_prob:
                    max_tr_prob = tr_prob
                    prev_st_selected = prev_st

            max_prob = max_tr_prob

            viterbi_dp[ind][each_tag] = {"prob": max_prob, "prev": prev_st_selected }
    max_prob = float(-1)
    best_state = None
    for state, data in viterbi_dp[-1].items():
        if data["prob"] > max_prob:
            max_prob = data["prob"]
            best_state = state

    optimal_tags.append(best_state)
    previous = best_state

    for k in range(len(viterbi_dp)-2, -1, -1):
        optimal_tags.insert(0, viterbi_dp[k+1][previous]["prev"])
        previous = viterbi_dp[k+1][previous]["prev"]

    return optimal_tags
```

This function calculates the viterbi decoding algorithm accuracy on dev data

In [36]:

```
def viterbi_decoding_accuracy(sentences):
    true_cnt = 0
    total_cnt = 0

    for i,sentence in enumerate(sentences):
        viterbi_pred = np.array(viterbi_decoding_algorithm(sentence))
        true_pred = dev_tags_list[i]
        true_cnt += sum(viterbi_pred == true_pred)
        total_cnt += len(dev_tags_list[i])

    viterbi_decode_accuracy = true_cnt/total_cnt
    viterbi_decode_accuracy = viterbi_decode_accuracy * 100
    return viterbi_decode_accuracy
```

In [37]:

```
print(viterbi_decoding_accuracy(dev_sentences_list))
```

```
94.64058041406108
```

Now we find the predictions of part-of-speech tags using viterbi decode algorithm on the test data

In [38]:

```
viterbi_tags_pred = []
for i,sentence in enumerate(test_sentences_list):
    viterbi_tags_pred += viterbi_decoding_algorithm(sentence)
df_test['viterbi_tags'] = viterbi_tags_pred
```

We now write the output predictions in a file named viterbi.out, in the same format of training data.

In [39]:

```
with open('viterbi.out','w') as f:
    for i,j in enumerate(df_test.values):
        line = str(j[0])+"\t"+str(j[1])+"\t"+str(j[3])
        if j[0]==1 and i!=0:
            f.write("\n")
        f.write(line)
        f.write('\n')
```

References

https://towardsdatascience.com/the-three-decoding-methods-for-nlp-23ca59cb1e9d (https://towardsdatascience.com/the-three-decoding-methods-for-nlp-23ca59cb1e9d)
https://medium.com/@jessica_lopez/understanding-greedy-search-and-beam-search-98c1e3cd821d (https://medium.com/@jessica_lopez/understanding-greedy-search-and-beam-search-98c1e3cd821d)
https://en.wikipedia.org/wiki/Viterbi_algorithm (https://en.wikipedia.org/wiki/Viterbi_algorithm)

THANK YOU