CSCI544 HW2 Report

1. Task 1: Vocabulary Creation
    - The selected threshold for unknown word replacement is 3(<3).
    - The total size of vocabulary before replacement is 43193.
    - The total size of vocabulary after replacement is 16920.
    - The total occurrences of the special token '<unk>' after replacement is 32537.
2. Task 2: Model Learning
    - I created two functions to calculate transition and emission.
    - number of **transition** parameters in HMM is 1392
    - number of **emission** parameters in HMM is 23373
    - For the transition dictionary, I add a 'start' token to calculate the probability of the word which is at the beginning of the sentence. I also make sure that there is no word 'start' in the training dataset.
    - Transition function may take some time (15 minutes).
    - In hmm.json, I separate tag and tags with space and word and tag with space(i.e. (s s'): value; (s x): value)
3. Task 3 Greedy Decoding with HMM
    - I wrote a find_tag_emm function to find the tag with corresponding emission given word. Then used for loops to predict every tag based on the previous tag.
    - The greedy decoding may take about >10 minutes to run on my computer.
    - The accuracy on the dev data (greedy decoding) is 92.5%
4. Task4 Viterbi Decoding with HMM
    - I transformed all transitions and emissions into matrices to speed up the algorithm.
    - I split the training data into lists of sentences and run the Viiterbi algorithm on every sentence to find the maximum prob to output the best path.
    - The Viterbi decoding may take about >10 minutes to run.
    - The accuracy on the dev data(Viterbi decoding) is 92.7%

Conclusion: Viterbi Decoding has higher accuracy than greedy decoding about 0.2%.

CSCI544_HW2 (1)

February 5, 2023

[1]:
```
import pandas as pd
import numpy as np
```

0.1 Task1: Vocabulary Creation

[151]:
```
## reading training text
f = open('train', "r")
text = [line.strip() for line in f.readlines()]
```

[138]:
```
text[:20]
```

[138]:
```
['1\tPierre\tNNP',
 '2\tVinken\tNNP',
 '3\t,\t,',
 '4\t61\tCD',
 '5\tyears\tNNS',
 '6\told\tJJ',
 '7\t,\t,',
 '8\twill\tMD',
 '9\tjoin\tVB',
 '10\tthe\tDT',
 '11\tboard\tNN',
 '12\tas\tIN',
 '13\ta\tDT',
 '14\tnonexecutive\tJJ',
 '15\tdirector\tNN',
 '16\tNov.\tNNP',
 '17\t29\tCD',
 '18\t.\t.',
 '',
 '1\tMr.\tNNP']
```

[152]:
```
## split each word and its tag into list
split = [item.split('\t') for item in text]
```

[6]:
```
split[:20]
```

[6]:
```
[['1', 'Pierre', 'NNP'],
 ['2', 'Vinken', 'NNP'],
 ['3', ',', ','],
 ['4', '61', 'CD'],
 ['5', 'years', 'NNS'],
 ['6', 'old', 'JJ'],
 ['7', ',', ','],
 ['8', 'will', 'MD'],
 ['9', 'join', 'VB'],
 ['10', 'the', 'DT'],
 ['11', 'board', 'NN'],
 ['12', 'as', 'IN'],
 ['13', 'a', 'DT'],
 ['14', 'nonexecutive', 'JJ'],
 ['15', 'director', 'NN'],
 ['16', 'Nov.', 'NNP'],
 ['17', '29', 'CD'],
 ['18', '.', '.'],
 [''],
 ['1', 'Mr.', 'NNP']]
```

[505]:
```
## create a dictionary to count word occurences
dict1 = {}
for item in split:
    if len(item) > 1:
        key = item[1]
    if key not in dict1:
        dict1[key] = 1
    else:
        dict1[key] += 1
```

[10]:
```
## sort dictionary in descending order
dict_sorted = {k: v for k, v in sorted(dict1.items(), key=lambda item: item[1],␣
 ↪reverse = True)}

## replace rare words whose occurence less than 3 with a special token '< unk␣
 ↪>'.
rare_words_occ = 0
rare_words = []
for k,v in dict_sorted.items():
    if v < 3:
        rare_words_occ += v
        rare_words.append([k,v])

## delete rare words and add unkown token to vocabulary and update vocabulary
for item in rare_words:
      dict_sorted.pop(item[0])
```

```
updict = {'<unk>': rare_words_occ}
dict_vocab = {**updict, **dict_sorted}
# dict_sorted.update({'<unk>': rare_words_occ})
# print('size of the vocabulary is', len(dict_vocab))
```

[11]:
```
## not removing puntuations and transform the dictionary into correct vocab form
i = 0
vocab = []
for k, v in dict_vocab.items():
    word = k + ' \t ' + str(i) + ' \t ' + str(v)
    vocab.append(word)
    i += 1
# print('total size of vocabulary is', len(vocab))
```

[12]:
```
## output the vocabulary into a txt file named vocab.txt
file = open('vocab.txt','w')
for item in vocab:
        file.write(item+"\n")
file.close()
```

[13]:
```
print('The selected threshold for unknown words replacement is', 3)
print('The total size of vocabulary is', len(vocab))
print('The total occurrences of the special token '< unk >' after replacement␣
 ↪is', rare_words_occ)
```

```
The selected threshold for unknown words replacement is 3
The total size of vocabulary is 16920
The total occurrences of the special token '< unk >' after replacement is 32537
```

0.2 Task 2: Model Learning

[14]:
```
## get unique tags from training dataset
tags1 = set([x[2] for x in split if len(x) > 1])
# len(tags1)
```

[14]:
```
45
```

[15]:
```
## get unique words from training dataset
words1 = set([x[1] for x in split if len(x) > 1])
# len(words1)
```

[15]:
```
43193
```

[84]:
```
## unknown words list
unk_word = [x[0] for x in rare_words]
```

[58]:
```
## create a dataframe for transition
tags = []
```

```
for item in tags1:
  tags.append(item)
row = tags.insert(0, 'start')
df = pd.DataFrame(columns=tags,
                  index=tags)
df_tran = df.drop(df.columns[0],axis=1)

## create a dataframe for emission
words = []
for item in words1:
  words.append(item)
df = pd.DataFrame(columns=words,
                  index=tags)
df_emm = df.drop(index='start')
```

[935]:
```
## transition function
def calculate_transition(current, next, training):
    denominator = 0
    count = 0
    if current == 'start':
        # count_current = 0
        for item in training:
            if len(item) > 1:
                if item[2] == next:
                    denominator += 1
                    if item[0] == '1':
                        count += 1
    else:
        for i in range(len(training)):
            if len(training[i]) > 1:
                if training[i][2] == current:
                    denominator += 1
                    index = i + 1
                    if (index < len(training)) and (len(training[index])>1):
                        if training[index][2] == next:
                            count += 1
        i += 1
    return count/denominator


## emission funtion
def calculate_emission(word, tag, training):
    denominator = 0
    count = 0
    for item in training:
        if (len(item) > 1) and (item[2] == tag):
            denominator += 1
```

```
            if item[1] == word:
#         if (len(item) > 1) and (item[1] == word) and (item[2] == tag):
                count += 1
    return [count/denominator, count, denominator]
```

[943]:
```
# calculate_emission('THE', 'DT', split)
```

[943]:
```
[0.00041891463027610284, 33, 78775]
```

[1012]:
```
split2_word_tag, split2_tag = [], []
for item in split:
    if len(item) < 2:
        split2_tag.append('')
    else:
        split2_tag.append(item[2])

for item in in_vocab:
    if len(item) < 2:
        split2_word_tag.append('')
    else:
        split2_word_tag.append(item[1] + ' '+item[2])
# len(split2_word_tag)
```

[1012]:
```
879558
```

[1014]:
```
from collections import defaultdict
dct = defaultdict(int)

for key in split2_word_tag:
    dct[key] += 1
# dct
```

[1014]:
```
defaultdict(int,
            {'Pierre NNP': 6,
             ', ,': 46476,
             '61 CD': 25,
             'years NNS': 1130,
             'old JJ': 213,
             'will MD': 2962,
             'join VB': 40,
             'the DT': 39517,
             'board NN': 297,
             'as IN': 3354,
             'a DT': 18445,
             'nonexecutive JJ': 6,
             'director NN': 309,
             'Nov. NNP': 234,
```

[233]:
```
# calculate_transition('start','WP$', split)
```

[233]: 0.006024096385542169

[115]:
```
# calculate_emission('PLC', 'NNP', test)
```

[115]: 0.07142857142857142

[286]:
```
## built transition dictionary: {(s, s'): t(s'|s)}
transition_dict = {}
for i in df_tran.index:
    for j in df_tran.columns:
        prob = calculate_transition(i, j, split)
        transition_dict[(i,j)] = prob
        df_tran.at[i,j] = prob
```

[131]:
```
## separate unknown words and words that are in vocab
## i will find emission for unknown words based on unk/states
in_vocab = []
notin_vocab = []
for item in split:
    if len(item) > 1 and item[1] in dict_sorted:
        in_vocab.append(item)
    else:
        notin_vocab.append(item)
```

[163]:
```
## count tags and tags with words functions
def calculate_denominator(tag, training):
    denominator = 0
    for item in training:
        if (len(item) > 1) and (item[2] == tag):
            denominator += 1
    return denominator
def calculate_count(word, tag, training):
    count = 0
    for item in training:
        if (len(item) > 1) and (item[1] == word) and (item[2] == tag):
            count += 1
    return count
```

[164]:
```
# calculate_count('.', '.', split)
```

[164]: 37452

[158]:
```
# calculate_denominator('NNP', split)
```

[158]: 87608

[961]:
```
d = {}
for tag in list(tags1):
    d[tag] = split2_tag.count(tag)
```

[1033]:
```
## build emission dictionary: {(s,x): e(x|s)}
## when the word is unknown, calculate e(s|'unk')
start = time.time()
emission_dict2 = {}
z = 0
for key, value in dict_vocab.items():
    start = time.time()
    for item in list(tags1):
#         start = time.time()
        word = key
        tag = item
        if (tag, word) in emission_dict2:
            continue
        prob = dct[word + ' ' + tag] / d[tag]
        #calculate_emission(word, tag, split)
        if prob != 0:
            emission_dict2[(tag,word)] = prob
        else:
            continue
    end = time.time()

    z += 1

end = time.time()
difference = end-start

# print("Time taken in seconds: ", difference)
```

```
Time taken in seconds:  0.00015091896057128906
```

[1040]:
```
dict_tag = {}
for item in list(tags1):
    val = calculate_denominator(item, split)
    dict_tag[item] = val
for item in notin_vocab:
    if len(item) > 1:
        tag = item[2]
        word = item[1]
        if (tag, 'unk') in emission_dict2:
            continue
        else:
            prob = calculate_denominator(tag, notin_vocab)/dict_tag[tag]
            emission_dict2[(tag,'unk')] = prob
```

[1042]:
```
# emission_dict2 == emission_dict
```

[1042]:
```
True
```

[ ]:
```
## build emission dictionary: {(s,x): e(x|s)}
## when the word is unknown, calculate e(s|'unk')
# emission_dict = {}
# for item in in_vocab:
#     word = item[1]
#     tag = item[2]
#     if (tag, word) in emission_dict:
#         continue
#     prob = calculate_count(word, tag, in_vocab)/calculate_denominator(tag,␣
 ↪split)
#     emission_dict[(tag,word)] = prob
```

[206]:
```
## adding unknown word to emission dictionary
# dict_tag = {}
# for item in list(tags1):
#     val = calculate_denominator(item, split)
#     dict_tag[item] = val
# for item in notin_vocab:
#     if len(item) > 1:
#         tag = item[2]
#         word = item[1]
#         if (tag, 'unk') in emission_dict:
#             continue
#         else:
#             prob = calculate_denominator(tag, notin_vocab)/dict_tag[tag]
#             emission_dict[(tag,'unk')] = prob
```

[ ]:
```

```

[ ]:
```

```

[1046]:
```
## turn tuple keys of dictionaries into keys and convert them to json form
dicts_tran = {" ".join(key): value for key, value in transition_dict.items()}
dicts_emm = {" ".join(key): value for key, value in emission_dict2.items()}
```

[360]:
```
a = {'transition': dicts_tran, 'emission': dicts_emm}
import json
json_file = json.dumps(a)
```

[361]:
```
## output json file
with open('hmm.json', 'w') as f:
    json.dump(json_file, f)
```

[274]:
```
# f = open ('hmm.json', "r")

# # Reading from file
# data = json.loads(f.read())
```

[362]:
```
## numbers of transition and emission parameters in my HMM
num_tran, num_emm = 0, 0
for key, value in transition_dict.items():
    if value != 0:
        num_tran += 1
for key,value in emission_dict.items():
    if value != 0:
        num_emm += 1
print('number of transition parameters in HMM is ', num_tran)
print('number of emission parameters in HMM is ', num_emm)
```

```
number of transition parameters in HMM is  1392
number of emission parameters in HMM is  23373
```

[ ]:

0.3 Task 3: Greedy Decoding withHMM

[363]:
```
f = open ('hmm.json', "r")

# Reading from file
data = json.loads(f.read())
tran = json.loads(data)['transition']
emm = json.loads(data)['emission']
```

[1047]:
```
# dicts_emm == emm
```

[1047]:
```
True
```

[1048]:
```
# dicts_tran == tran
```

[1048]:
```
True
```

[1049]:
```
emm = dicts_emm
tran = dicts_tran
```

[365]:
```
f_dev = open('dev', "r")
text_dev = [line.strip() for line in f_dev.readlines()]
split_dev = [item.split('\t') for item in text_dev]
```

[ ]:

[ ]:
```
## greedy algorithm

def find_max_tag_emm(x):
    values, keys = [], []
    for key, value in emm.items():
        if key.split(' ')[1] == x:
            values.append(value)
            keys.append(key.split(' ')[0])
    if len(values) == 0:
        for key, value in emm.items():
            if key.split(' ')[1] == 'unk':
                values.append(value)
                keys.append(key.split(' ')[0])
#     return keys[np.argmax(values)].split(' ')[0]
    return keys,values

predicted_tags, previous = [],''
i = 0
for item in split_dev:
    if len(item) < 2:
        continue
    word = item[1]
    if item[0] == '1':
        previous = 'start'
    tran_find = df_tran.loc[previous]
    emm_find = find_max_tag_emm(word)
    tags = emm_find[0]
    pred = tags[np.argmax(tran_find[emm_find[0]] * emm_find[1])]
    predicted_tags.append(pred)
    previous = pred
    i += 1
#     print(i)
```

[470]:
```
actual_tags = [item[2] for item in split_dev if len(item) > 1]
acc = 0
for i in range(len(actual_tags)):
    if predicted_tags[i] == actual_tags[i]:
        acc += 1
accuracy = acc/len(actual_tags)
# accuracy
```

[470]:
```
0.9254219537368709
```

[875]:
```
print('The accuracy on the dev data(greedy decoding) is ', accuracy)
```

```
The accuracy on the dev data(greedy decoding) is  0.9254219537368709
```

[471]:
```
## use test data

f_test = open('test', "r")
text_test = [line.strip() for line in f_test.readlines()]
split_test = [item.split('\t') for item in text_test]
```

[473]:
```
# len(split_test)
```

[473]:
```
135115
```

[ ]:
```
## greedy algorithm on test data

predicted_tags, previous = [],''
i = 0
for item in split_test:
    if len(item) < 2:
        predicted_tags.append('')
        continue
    word = item[1]
    if item[0] == '1':
        previous = 'start'
    tran_find = df_tran.loc[previous]
    emm_find = find_max_tag_emm(word)
    tags = emm_find[0]
    pred = tags[np.argmax(tran_find[emm_find[0]] * emm_find[1])]
    predicted_tags.append(pred)
    previous = pred
    i += 1
#     print(i)
```

[503]:
```
##combine tags with text in test data and output as 'greedy.out'
output_test_pred = []
for i in range(len(text_test)):
    if text_test[i] != '':
        output_test_pred.append(text_test[i] + '\t'+ predicted_tags[i])
    else:
        output_test_pred.append(text_test[i])
```

[508]:
```
file_test_pred = open('greedy.out','w')
for item in output_test_pred:
        file_test_pred.write(item+"\n")
file_test_pred.close()
```

[ ]:
```

```

0.4 Task 4: Viterbi Decoding with HMM

[635]:
```
def find_emm_prob(tag, word):
    key = tag + ' ' + word
    if key in emm:
        return emm[key]
    elif (tag+' ' + 'unk' in emm) and word not in vocab_lists:
        return emm[tag+' '+ 'unk']
    return 0
```

[ ]:
```
words_em = []
for key, value in emission_dict.items():
    words_em.append(key[1])
df_emm2= pd.DataFrame(columns=list(set(words_em)),
                  index=list(set(tags1)))
for key, value in emission_dict.items():
    df_emm2.at[key[0], key[1]] = value
df_emm2.fillna(0, inplace = True)
# df_emm2
```

[751]:
```
## source: https://towardsdatascience.com/implementing-part-of-
##␣
 ↪speech-tagging-for-english-words-using-viterbi-algorithm-from-scratch-9ded56b29133
def viterbi_alog(sentence, tags):
    tags = set(list(tags))
    path = {}
    for t in tags:
        if sentence[0] not in df_emm2.columns:
            path[t, 0] = df_tran.loc['start', t] * df_emm2.loc[t, 'unk']
        else:
            path[t, 0] = df_tran.loc['start', t] * df_emm2.loc[t, sentence[0]]

    for i in range(1, len(sentence)):
        if sentence[i] not in df_emm2.columns:
            obs = 'unk'
        else:
            obs = sentence[i]
        for t in tags:
            v1 = [(path[k, i-1] * df_tran.loc[k, t] * df_emm2.loc[t, obs], k)␣
 ↪for k in tags]
            k = sorted(v1)[-1][1]
            path[t, i] = path[k, i-1]* df_tran.loc[k,t] * df_emm2.loc[t, obs]

    best = []
    for i in range(len(sentence) - 1, -1, -1):
        k = sorted([(path[k, i], k) for k in tags])[-1][1]
        best.append((sentence[i], k))
```

```
    best.reverse()

    return [str(item[0]) + " " + str(item[1]) for item in best]
```

[855]:
```
dev_copy = split_dev.copy()
dev_copy.append([''])
```

[856]:
```
split_in_sentence = []
sentence = []
for item in dev_copy:
    if len(item) < 2:
        split_in_sentence.append(sentence)
        sentence = []
    else:
        sentence.append(item[1])
```

[ ]:
```
start = time.time()
predicted_tags_viterbi = []
z = 1
for item in split_in_sentence:
    predicted_tags_viterbi.extend(viterbi_alog(item, tags1))
    predicted_tags_viterbi.append('')
    z += 1
    print(z)
end = time.time()
difference = end-start
# print("Time taken in seconds: ", difference)
```

[871]:
```
predicted_t_dev = []
for item in predicted_tags_viterbi:
    a = item.split(' ')
    if len(a) > 1:
        predicted_t_dev.append(a[1])
```

[874]:
```
# actual_tags = [item[2] for item in split_dev if len(item) > 1]
predicted_tags_viterbi2 = [item for item in predicted_t_dev if item != '']
acc_viterbi = 0
for i in range(len(actual_tags)):
    if predicted_tags_viterbi2[i] == '':
        continue
    if predicted_tags_viterbi2[i] == actual_tags[i]:
        acc_viterbi += 1
accuracy_viterbi = acc_viterbi/len(actual_tags)
print('The accuracy on the dev data(Viterbi decoding) is ', accuracy_viterbi)
```

```
The accuracy on the dev data(Viterbi algorithm) is  0.9272357476777366
```

[ ]:

[ ]:
```
## output predcitions for test data
```

[877]:
```
test_copy = split_test.copy()
test_copy.append([''])

split_in_sentence2 = []
sentence2 = []
for item in test_copy:
    if len(item) < 2:
        split_in_sentence2.append(sentence2)
        sentence2 = []
    else:
        sentence2.append(item[1])
```

[ ]:
```
start = time.time()
predicted_tags_viterbi_test = []
z = 1
for item in split_in_sentence2:
    predicted_tags_viterbi_test.extend(viterbi_alog(item, tags1))
    predicted_tags_viterbi_test.append('')
    z += 1
    print(z)
end = time.time()
difference = end-start
# print("Time taken in seconds: ", difference)
```

[891]:
```
predicted_t_test = []
for item in predicted_tags_viterbi_test:
    a = item.split(' ')
    if len(a) > 1:
        predicted_t_test.append(a[1])
```

[897]:
```
predicted_t_test = []
for item in predicted_tags_viterbi_test:
    a = item.split(' ')
    if len(a) > 1:
        predicted_t_test.append(a[1])
    else:
        predicted_t_test.append(item)
```

[902]:
```
##combine tags with text in test data and output as 'greedy.out'
output_test_pred_viterbi = []
for i in range(len(text_test)):
    if text_test[i] != '':
```

```
        output_test_pred_viterbi.append(text_test[i] + '\t'+␣
↪predicted_t_test[i])
    else:
        output_test_pred_viterbi.append(text_test[i])
```

[907]:
```
file_test_pred_viterbi = open('viterbi.out','w')
for item in output_test_pred_viterbi:
        file_test_pred_viterbi.write(item+"\n")
file_test_pred_viterbi.close()
```

[ ]:
```

```