CSCI-544-HW3

October 21, 2021

Name: Nidhi Rajkumar Saini
USC ID: 3730422872

[ ]:
```
import pandas as pd
from collections import Counter
import json
import numpy as np
```

1 Vocabulary Creation

The first task is to create a vocabulary using the training data. In HMM, one important problem when creating the vocabulary is to handle unknown words. One simple solution is to replace rare words whose occurrences are less than a threshold (e.g. 3) with a special token '< unk >'.

- Used pandas read_csv method to read the train, dev and test dataset into a pandas dataframe using tab separator.
- pandas.DataFrame.head method returns the first n rows of the dataframe.

[ ]:
```
train = pd.read_csv('data/train', sep = '\t', header = None, names =␣
 ↪['index','word','pos'])
train.head(10)
```

[ ]:

	index	word	pos
0	1	Pierre	NNP
1	2	Vinken	NNP
2	3	,	,
3	4	61	CD
4	5	years	NNS
5	6	old	JJ
6	7	,	,
7	8	will	MD
8	9	join	VB
9	10	the	DT

[ ]:
```
dev = pd.read_csv('data/dev', sep = '\t', header = None, names =␣
 ↪['index','word','pos'])
dev.head(10)
```

[ ]:

	index	word	pos
0	1	The	DT
1	2	Arizona	NNP
2	3	Corporations	NNP
3	4	Commission	NNP
4	5	authorized	VBD
5	6	an	DT
6	7	11.5	CD
7	8	%	NN
8	9	rate	NN
9	10	increase	NN

[ ]:
```
test = pd.read_csv('data/test',sep='\t',header=None,names=['index','word'])
test.head(10)
```

[ ]:

	index	word
0	1	Influential
1	2	members
2	3	of
3	4	the
4	5	House
5	6	Ways
6	7	and
7	8	Means
8	9	Committee
9	10	introduced

1.1 What is the selected threshold for unknown words replacement?

- I set the threshold to 2 to replace words having frequency less than threshold with the < unk > token.
- Used Counter class to get the frequency of all words in the training set and converted Counter object to dictionary.
- Replaced all the rarely occuring words with < unk > token and updated count of < unk > token in the dictionary.

[ ]:
```
THRESHOLD = 2
word_list = train['word'].tolist()
word_counter = Counter(word_list)
vocabulary_dict = dict(word_counter)
temp_dict = dict(word_counter)
for word in temp_dict:
    if(vocabulary_dict[word] < THRESHOLD):
        if('<unk>' not in vocabulary_dict):
            vocabulary_dict['<unk>'] = 0
        vocabulary_dict['<unk>'] += vocabulary_dict[word]
        del vocabulary_dict[word]
```

1.2 What is the total size of your vocabulary and what is the total occurrences of the special token '< unk >' after replacement?

- There are 23183 unique words in the vocabulary including the < unk > token.
- There are 20011 occurences of < unk > token in the data after replacement.

[ ]:
```
len(vocabulary_dict)
```

[ ]:
```
23183
```

[ ]:
```
vocabulary_dict['<unk>']
```

[ ]:
```
20011
```

1.3 Created a vocabulary using the training data in the file train and outputted the vocabulary into a txt file named vocab.txt.

[ ]:
```
vocab_list = []
unk_count = vocabulary_dict['<unk>']
del vocabulary_dict['<unk>']
for word in vocabulary_dict:
    vocab_list.append((vocabulary_dict[word],word))
vocab_list.sort(reverse = True)
vocab_list[:10]
```

[ ]:
```
[(46476, ','),
 (39533, 'the'),
 (37452, '.'),
 (22104, 'of'),
 (21305, 'to'),
 (18469, 'a'),
 (15346, 'and'),
 (14609, 'in'),
 (8872, "'s"),
 (7743, 'for')]
```

[ ]:
```
with open('vocab.txt','w') as file:
    lines = ['<unk>\t0\t'+ str(unk_count)+'\n']
    idx = 1
    for each in vocab_list:
        line = each[1] + '\t' + str(idx) + '\t' + str(each[0])+'\n'
        lines.append(line)
        idx += 1
    file.writelines(lines)
```

2 Model Learning - HMM

The second task is to learn an HMM from the training data. Remember that the solution of the emission and transition parameters in HMM are in the following formulation: $t(s0 \mid s) =$

count(ss0)/count(s) e(x | s) = count(sx)/count(s) where t(ů | ů) is the transition parameter and e(ů | ů) is the emission parameter.

- Created a dictionary from a Counter object having tag names and corresponding frequency of the tag in the training data.
- This dictionary gives us the count(s) term in the above formulaes.

[ ]:
```
print('Learning HMM Probabilities...')
```

```
Learning HMM Probabilities...
```

[ ]:
```
tag_counter = Counter(train['pos'])
tag_dict = dict(tag_counter)
list(tag_dict.items())[:10]
```

[ ]:
```
[('NNP', 87608),
 (',', 46480),
 ('CD', 34876),
 ('NNS', 57859),
 ('JJ', 58944),
 ('MD', 9437),
 ('VB', 25489),
 ('DT', 78775),
 ('NN', 127534),
 ('IN', 94758)]
```

- The next code block is equivalent to creating a beginning of sentence tag at the start of each sentence.
- The code counts the number of times each tag appears as the first tag in a sentence and divides it by the number of sentences, thus, giving us the probability of the tag occuring at the start of the sentence.

[ ]:
```
first_tag_probs = {}
count_first_tag = {}
for tag in train[train['index']==1]['pos'].tolist():
    if tag not in count_first_tag:
        count_first_tag[tag] = 0
    count_first_tag[tag] += 1

for key in count_first_tag:
    first_tag_probs[key] = count_first_tag[key]/
 ↪len(train[train['index']==1]['pos'].tolist())
```

[ ]:
```
list(first_tag_probs.items())[:10]
```

[ ]:
```
[('NNP', 0.19789104610393007),
 ('DT', 0.21911141347009264),
 ('IN', 0.1288398137003506),
 ('PRP', 0.06148935056779528),
```

```
('EX', 0.004238840337013972),
('``', 0.07472918520069077),
('CD', 0.011225077188759224),
('RBR', 0.0020932544874143074),
('NNS', 0.041237113402061855),
('NN', 0.0411847820398765)]
```

- Creating a list of list 'sentences' containing each sentence of the training set inside a separate list, where all words of the sentence which are not found in the vocabulary are replaced by < unk > token.
- Also, created a similar list of lists 'sentence_tags' which contains the corresponding tags in similar structure.
- Repeated this process for train, dev and test (no sentence_tags in test) datasets.

[ ]:
```
sentences = []
sentence_tags = []
tags = []
sentence = None
for row in train.values.tolist():
    if row[0] == 1:
        if sentence:
            sentence_tags.append(tags)
            sentences.append(sentence)
        sentence = []
        tags = []
    if row[1] not in vocabulary_dict:
        sentence.append('<unk>')
    else:
        sentence.append(row[1])
    tags.append(row[2])
sentence_tags.append(tags)
sentences.append(sentence)
```

[ ]:
```
sentences[:1]
```

[ ]:
```
[['Pierre',
  'Vinken',
  ',',
  '61',
  'years',
  'old',
  ',',
  'will',
  'join',
  'the',
  'board',
  'as',
  'a',
```

```
  'nonexecutive',
  'director',
  'Nov.',
  '29',
  '.']]
```

[ ]:
```
sentence_tags[:1]
```

[ ]:
```
[['NNP',
  'NNP',
  ',',
  'CD',
  'NNS',
  'JJ',
  ',',
  'MD',
  'VB',
  'DT',
  'NN',
  'IN',
  'DT',
  'JJ',
  'NN',
  'NNP',
  'CD',
  '.']]
```

[ ]:
```
dev_sentences = []
dev_sentence_tags = []
tags = []
sentence = None
for row in dev.values.tolist():
    if row[0] == 1:
        if sentence:
            dev_sentence_tags.append(tags)
            dev_sentences.append(sentence)
        sentence = []
        tags = []
    if row[1] not in vocabulary_dict:
        sentence.append('<unk>')
    else:
        sentence.append(row[1])
    tags.append(row[2])
dev_sentence_tags.append(tags)
dev_sentences.append(sentence)
```

[ ]:
```
test_sentences = []
sentence = None
for row in test.values.tolist():
```

```
    if row[0] == 1:
        if sentence:
            test_sentences.append(sentence)
        sentence = []
    if row[1] not in vocabulary_dict:
        sentence.append('<unk>')
    else:
        sentence.append(row[1])
test_sentences.append(sentence)
```

- Created function create_bigram_dict to compute the count of s->s' using training data variable 'sentence_tags', returns a dictionary with key structured as a tuple (s,s') where s is the tag and s' is the next tag in the sentence.

[ ]:
```
def create_bigram_dict(sentences):
    bigram_dict = {}
    for sentence in sentences:
        for idx, word in enumerate(sentence[:-1]):
            key = (word, sentence[idx+1])
            if key not in bigram_dict:
                bigram_dict[key] = 0
            bigram_dict[key]+=1
    return bigram_dict
```

[ ]:
```
bigram_tag_dict = create_bigram_dict(sentence_tags)
```

[ ]:
```
list(bigram_tag_dict.items())[:10]
```

[ ]:
```
[(('NNP', 'NNP'), 33139),
 (('NNP', ','), 12131),
 ((',', 'CD'), 987),
 (('CD', 'NNS'), 5502),
 (('NNS', 'JJ'), 995),
 (('JJ', ','), 1717),
 ((',', 'MD'), 490),
 (('MD', 'VB'), 7541),
 (('VB', 'DT'), 5661),
 (('DT', 'NN'), 37299)]
```

- Calculated the transition probabilities using the bigram_tag_dict and tag_dict created above. The resulting dictionary contains keys similar to bigram_tag_dict i.e., in tuple format (s,s') with the corresponding transition probability as value.

[ ]:
```
transition_probs = {}
for key in bigram_tag_dict:
    transition_probs[key] = (bigram_tag_dict[key])/(tag_dict[key[0]])
```

[ ]:
```
list(transition_probs.items())[:10]
```

[ ]:
```
[(('NNP', 'NNP'), 0.3782645420509543),
 (('NNP', ','), 0.13846908958086018),
 ((',', 'CD'), 0.021234939759036144),
 (('CD', 'NNS'), 0.15775891730703062),
 (('NNS', 'JJ'), 0.017196978862406887),
 (('JJ', ','), 0.029129343105320303),
 ((',', 'MD'), 0.010542168674698794),
 (('MD', 'VB'), 0.7990886934407121),
 (('VB', 'DT'), 0.22209580603397544),
 (('DT', 'NN'), 0.4734877816566169)]
```

- Created tag_word_dict, a dictionary with key structured as a tuple (s,x), which contains the count of s->x using training data variables 'sentences' and 'sentence_tags' where s is the tag and x is the corresponding word in the sentence.

[ ]:
```
tag_word_dict = {}
for i in range(len(sentences)):
    for j in range(len(sentences[i])):
        key = (sentence_tags[i][j], sentences[i][j])
        if key not in tag_word_dict:
            tag_word_dict[key] = 0
        tag_word_dict[key]+=1
list(tag_word_dict.items())[:10]
```

[ ]:
```
[(('NNP', 'Pierre'), 6),
 (('NNP', 'Vinken'), 2),
 ((',', ','), 46476),
 (('CD', '61'), 25),
 (('NNS', 'years'), 1130),
 (('JJ', 'old'), 213),
 (('MD', 'will'), 2962),
 (('VB', 'join'), 40),
 (('DT', 'the'), 39517),
 (('NN', 'board'), 297)]
```

- Calculated the emission probabilities using the tag_word_dict and tag_dict created above. The resulting dictionary contains keys similar to tag_word_dict i.e., in tuple format (s,x) with the corresponding emission probability as value.

[ ]:
```
emission_probs = {}
for key in tag_word_dict:
    emission_probs[key] = (tag_word_dict[key])/(tag_dict[key[0]])
list(emission_probs.items())[:10]
```

[ ]:
```
[(('NNP', 'Pierre'), 6.84868961738654e-05),
 (('NNP', 'Vinken'), 2.2828965391288468e-05),
 ((',', ','), 0.9999139414802065),
 (('CD', '61'), 0.0007168253240050465),
```

```
(('NNS', 'years'), 0.019530237301024905),
(('JJ', 'old'), 0.003613599348534202),
(('MD', 'will'), 0.3138709335593939),
(('VB', 'join'), 0.0015693044058221193),
(('DT', 'the'), 0.5016439225642653),
(('NN', 'board'), 0.0023287907538381922)]
```

- Created dictionary hmm_dict which consists of key 'transition' with the 'transition_probs' dictionary as value, and key 'emission' with the 'emission_probs' dictionary as value.

[ ]:
```
hmm_dict = {}
hmm_dict['transition'] = transition_probs
hmm_dict['emission'] = emission_probs
```

- Since json files cannot accept tuple as keys, created a copy of hmm_dict with keys of transition and emission probabilities as strings instead of tuples with elements of tuple separated by a |.

[ ]:
```
hmm_dict_copy = {'transition':{},'emission':{}}
for key in hmm_dict:
    for pair in hmm_dict[key]:
        hmm_dict_copy[key][pair[0]+'|'+pair[1]] = hmm_dict[key][pair]
```

- Wrote the hmm_dict_copy dictionary to hmm.json as asked by the problem statement.

[ ]:
```
with open('hmm.json','w+') as file:
    json.dump(hmm_dict_copy,file)
```

2.1 How many transition and emission parameters in your HMM?

- There are 1351 transition parameters in the transition probabilities dictionary.
- There are 30303 emission parameters in the emission probabilities dictionary.

[ ]:
```
len(transition_probs)
```

[ ]:
```
1351
```

[ ]:
```
len(emission_probs)
```

[ ]:
```
30303
```

3 Greedy Decoding with HMM

The third task is to implement the greedy decoding algorithm with HMM.

- Created a word-wise dictionary containing each word from vocabulary as key and value as a dictionary containing only the emission probabilities for that word.
- This is done to make greedy decoding run faster by iterating over only the corresponding emissions of a word.

[ ]:
```
emission_probs_2 = {}

for key in emission_probs:
    if key[1] not in emission_probs_2:
        emission_probs_2[key[1]] = {}
    emission_probs_2[key[1]][key]= emission_probs[key]
emission_probs_2['and']
```

[ ]:
```
{('CC', 'and'): 0.6722180830082833,
 ('JJ', 'and'): 5.0895765472312706e-05,
 ('IN', 'and'): 1.0553198674518247e-05,
 ('NN', 'and'): 1.5682092618439004e-05,
 ('NNP', 'and'): 2.2828965391288468e-05}
```

- Created function greedy_decode which takes a list of words as input (the words are strictly from the vocabulary otherwise < unk >).
- This function iterates over the words in the input sentence. If it is the first word, the emission probability for each (tag,word) pair in 'emission_probs_2[word]' is multiplied with probability of the tag being the first tag by using 'first_tag_probs' dictionary.
- If the word is not the first word, then, we calculate probability as the multiplication of the emission probability for each (tag,word) pair in 'emission_probs_2[word]' and the transition probability of the corresponding tag given the previous tag.
- At each iteration, we save the tag having the maximum probability and move to the next word. Hence this approach is greedy in nature.

[ ]:
```
def greedy_decode(sentence):
    max_keys = []
    for i,word in enumerate(sentence):
        max_prob = 0
        max_key = None
        if i == 0:
            for key in emission_probs_2[word]:
                em_prob = emission_probs_2[word][key]
                ft_prob = first_tag_probs.get(key[0],0)
                if em_prob*ft_prob >= max_prob:
                    max_prob = em_prob*ft_prob
                    max_key = key
        else:
            for key in emission_probs_2[word]:
                prev_emission = max_keys[-1][0]
                transition = (prev_emission,key[0])
                tr_prob = transition_probs.get(transition,0)
                total_prob = tr_prob * emission_probs_2[word][key]
                if total_prob >= max_prob:
                    max_prob = total_prob
                    max_key = key
        max_keys.append(max_key)
    return [x[0] for x in max_keys]
```

3.1 What is the accuracy of greedy decoding on the dev data?

The computed accuracy on the development data for my model comes out to be 93.516% as computed in Part 5 (Accuracy Computation) of the notebook.

4 Viterbi Decoding with HMM

- Created function 'make_log_probs' which takes a dictionary as input and converts all the probability values in the dictionary to log probabilities and outputs it as a dictionary with the same keys but values as log probabilities.
- Used this function to create log probabilities dictionaries for transition, emission and first-tag probabilities.

[ ]:
```
def make_log_probs(probs):
    return {key:np.log(probs[key]) for key in probs}

transition_log_probs = make_log_probs(transition_probs)
emission_log_probs = make_log_probs(emission_probs)
first_tag_log_probs = make_log_probs(first_tag_probs)
```

- Created a function 'viterbi_decode' which takes list of words as input and performs viterbi decoding using log of transition, emission and first-tag probabilities created above.
- Viterbi algorithm is a dynamic programming algorithm which returns an optimal sequence of tags for words of a given sentence.
- In this algorithm, we use a optimal substructure 'V' which stores a word-wise list of dictionaries with each dictionary having all possible tags in the dataset as keys and value as another dictionary having the probability of that tag and the name of the previous tag in the sequence being computed.
- For the first word, first tag and emission log probabilities are used to obtain probability of the tag with previous tag as None.
- For all other words, the log probability value of stored for all tags of previous word in V along with the transition and emission log probabilities are used to get probability of the tag for the current word and stored at corresponding position in V.
- Once V is calculated, we backtrack from the tag of the last word which has the maximum log probability, till the first tag using the previous tag key called 'prev' in V.
- The above algorithm gives a very small probability (log probability = -1000) to keys which are not found in the transition, emission and first-tag log probability dictionary

[ ]:
```
def viterbi_decode(sentence):
    tags = list(tag_dict.keys())

    K = len(tags)
    T = len(sentence)
    V = [{}]

    for tag in tags:
        if sentence[0] != '<unk>':
```

```
            V[0][tag] = {"prob":first_tag_log_probs.
↪get(tag,-1000)+emission_log_probs.get((tag,sentence[0]),-1000),
                    "prev" : None}
        else:
            V[0][tag] = {"prob":first_tag_log_probs.get(tag,-1000),"prev" :␣
↪None}

    for i in range(1,T):
        V.append({})
        for tag in tags:
            if sentence[i] != '<unk>':
                max_tr_prob = V[i - 1][tags[0]]["prob"] + transition_log_probs.
↪get((tags[0],tag),-1000)  + emission_log_probs.get((tag,sentence[i]),-1000)
            else:
                max_tr_prob = V[i - 1][tags[0]]["prob"] + transition_log_probs.
↪get((tags[0],tag),-1000)
            prev_st_selected = tags[0]
            for prev_st in tags[1:]:
                tr_prob = V[i-1][prev_st]['prob'] + transition_log_probs.
↪get((prev_st,tag),-1000) + emission_log_probs.get((tag,sentence[i]),-1000)
                if tr_prob > max_tr_prob:
                    max_tr_prob = tr_prob
                    prev_st_selected = prev_st

            max_prob = max_tr_prob
            V[i][tag] = {"prob":max_prob,"prev":prev_st_selected}

    opt = []
    max_prob = float('-inf')
    best_tag = None
    for tag, data in V[-1].items():
        if data['prob'] > max_prob:
            max_prob = data['prob']
            best_tag = tag
    opt.append(best_tag)
    previous = best_tag

    for i in range(len(V)-2,-1,-1):
        opt.insert(0,V[i+1][previous]['prev'])
        previous = V[i + 1][previous]["prev"]

    return opt
```

4.1 What is the accuracy of viterbi decoding on the dev data?

The computed accuracy on the development data for my model comes out to be 94.634% as computed in Part 5 (Accuracy Computation) of the notebook.

5 Accuracy Computation

- Created function compute_accuracy which computes the accuracy for both greedy and viterbi decoding.
- It counts the number of tags that the decoder predicted correctly in the entire given dataset and divides it by the total number of tags in the entire given dataset and prints the values in percentage

[ ]:
```
def compute_accuracy(sentences,sentence_tags):
    greedy_count = 0
    viterbi_count = 0
    actual_count = 0

    for i,sentence in enumerate(sentences):
        greedy_tags = np.array(greedy_decode(sentence))
        viterbi_tags = np.array(viterbi_decode(sentence))
        actual_tags = np.array(sentence_tags[i])
        greedy_count += np.sum(greedy_tags==actual_tags)
        viterbi_count += np.sum(viterbi_tags==actual_tags)
        actual_count += len(sentence_tags[i])

    greedy_acc = greedy_count/actual_count
    viterbi_acc = viterbi_count/actual_count

    print('Accuracy for Greedy Decoding: '+str(greedy_acc*100)+"%\nAccuracy for␣
 ↪Viterbi Decoding: "+str(viterbi_acc*100)+"%")
```

[ ]:
```
print('Computing Accuracy on dev...')
compute_accuracy(dev_sentences,dev_sentence_tags)
```

```
Computing Accuracy on dev...
Accuracy for Greedy Decoding: 93.51587638880457%
Accuracy for Viterbi Decoding: 94.63375022767288%
```

6 Testing on Test Data

Predicting the part-of-speech tags of the sentences in the test data and output the predictions in files named greedy.out (for greedy decoder) and viterbi.out (for viterbi decoder), in the same format of training data.

[ ]:
```
print('Computing tags on test data...')
predicted_test_greedy_tags = []
predicted_test_viterbi_tags = []
for sentence in test_sentences:
    predicted_test_greedy_tags += greedy_decode(sentence)
    predicted_test_viterbi_tags += viterbi_decode(sentence)
```

```
Computing tags on test data...
```

[ ]:
```
test['greedy_tags'] = predicted_test_greedy_tags
test['viterbi_tags'] = predicted_test_viterbi_tags
```

[ ]:
```
with open('greedy.out','w+') as f:
    test_list = test[['index','word','greedy_tags']].values.astype(str).tolist()
    lines = ['\t'.join(x)+'\n' for x in test_list]
    for i,line in enumerate(lines):
        if line[:2] == '1\t' and i!=0:
            f.write('\n')
        f.write(line)

with open('viterbi.out','w+') as f:
    test_list = test[['index','word','viterbi_tags']].values.astype(str).
 ↪tolist()
    lines = ['\t'.join(x)+'\n' for x in test_list]
    for i,line in enumerate(lines):
        if line[:2] == '1\t' and i!=0:
            f.write('\n')
        f.write(line)
```

[ ]:
```
print('Done!')
```

```
Done!
```

7 References

https://en.wikipedia.org/wiki/Viterbi_algorithm

[ ]:
```
!apt-get install texlive-xetex texlive-fonts-recommended␣
 ↪texlive-latex-recommended
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

[2]:
```
!jupyter nbconvert --Application.log_level=CRITICAL --to pdf "/content/drive/
 ↪MyDrive/Colab Notebooks/CSCI-544-HW3.ipynb"
```

[ ]:
```

```