CSCI544: Homework Assignment No. 2

Submitted by:

Name: Dhiraj Chaurasia
USC ID#: 1556515687

Task 1: Vocabulary Creation (20 points)

Steps:

1. Read the file
2. Count the occurrence of words in a temporary dictionary 'word_counter' first
3. Set a minimum threshold 'threshold'
4. If a word occurs less than 'threshold' times, remap it as '<unk>' in the final dictionary 'word_counter_final'
5. Write the dictionary to vocab.txt

```
# Task 1: Vocabulary Creation (20 points)
def create_vocabulary():
    # Read the file
    # Count the occurrence of words in a temporary dictionary 'word_counter' first
    with open('data/train') as file:
        train_data = []
        temp = []
        word_counter = Counter()
        s = file.read().splitlines()
        for i in range(len(s)):
            if s[i] == '':
                train_data.append(temp)
                temp = []
                continue
            one_line = s[i].split('\t')
            word_counter[one_line[1]] += 1
            temp.append(one_line)
        train_data.append(temp)

    # Set a minimum threshold 'threshold'
    # If a word occurs less than 'threshold' times, remap it as '<unk>' in the final dictionary 'word_counter_final'
    threshold = 2
    word_counter_final = Counter()
    for k, v in word_counter.items():
        if v <= threshold:
            word_counter_final['<unk>'] += v
        else:
            word_counter_final[k] = v

    # Write the dictionary to vocab.txt
    with open('vocab.txt', 'w') as file:fi
        file.write(f"<unk>\t0\t{word_counter_final['<unk>']}\n")
        word_counter_final.pop('<unk>', None)
        i = 1
        for word, freq in word_counter_final.most_common():
            s = f"{word}\t{i}\t{freq}\n"
            i += 1
            file.write(s)
    return word_counter_final, train_data
```

```
word_counter_final, train_data = create_vocabulary()
```

```
word_counter_final.most_common()
```

```
[(',', 46476),
 ('the', 39533),
 ('.', 37452),
 ('of', 22104),
 ('to', 21305),
 ('a', 18469),
 ('and', 15346),
 ('in', 14609),
 ("'s", 8872),
 ('for', 7743),
 ('that', 7723),
 ('$', 6762),
 ('is', 6735),
 ('``', 6673),
 ('The', 6578),
 ("''", 6500),
 ('said', 5418),
 ('on', 4905),
 ('%', 4718),
```

```
len(word_counter_final)
```

```
16919
```

```
<unk>   0       32537
,       1       46476
the     2       39533
.       3       37452
of      4       22104
to      5       21305
a       6       18469
and     7       15346
in      8       14609
's      9       8872
for     10      7743
that    11      7723
$       12      6762
is      13      6735
``      14      6673
The     15      6578
''      16      6500
said    17      5418
on      18      4905
%       19      4718
it      20      4509
by      21      4274
from    22      4238
at      23      4142
million 24      4122
as      25      4054
with    26      3987
Mr.     27      3856
are     28      3629
was     29      3615
```

Q1) What is the selected threshold for unknown words replacement?

- The selected threshold in my implementation is 2.

Q2) What is the total size of your vocabulary and what is the total occurrences of the special token ‘< unk >’ after replacement?

- The total size of my vocabulary is 16919 (excluding the special token <unk>) and 16920 including the special token <unk>.
- The total occurrences of the special token ‘<unk>’ after replacement is 32537.

Task 2: Model Learning (20 points)

Steps:

1. Get the emission and transition count first
2. Maintain a tag_counter that counts the number of occurrences of each tag
3. For the first word, use <start> as the previous token.
4. For the last word, we don't have to calculate transition_probabilities.
5. Use the tag_counter to normalize emission_probabilities and transition_probabilities
6. Write transition and emission probabilities to hmm.json

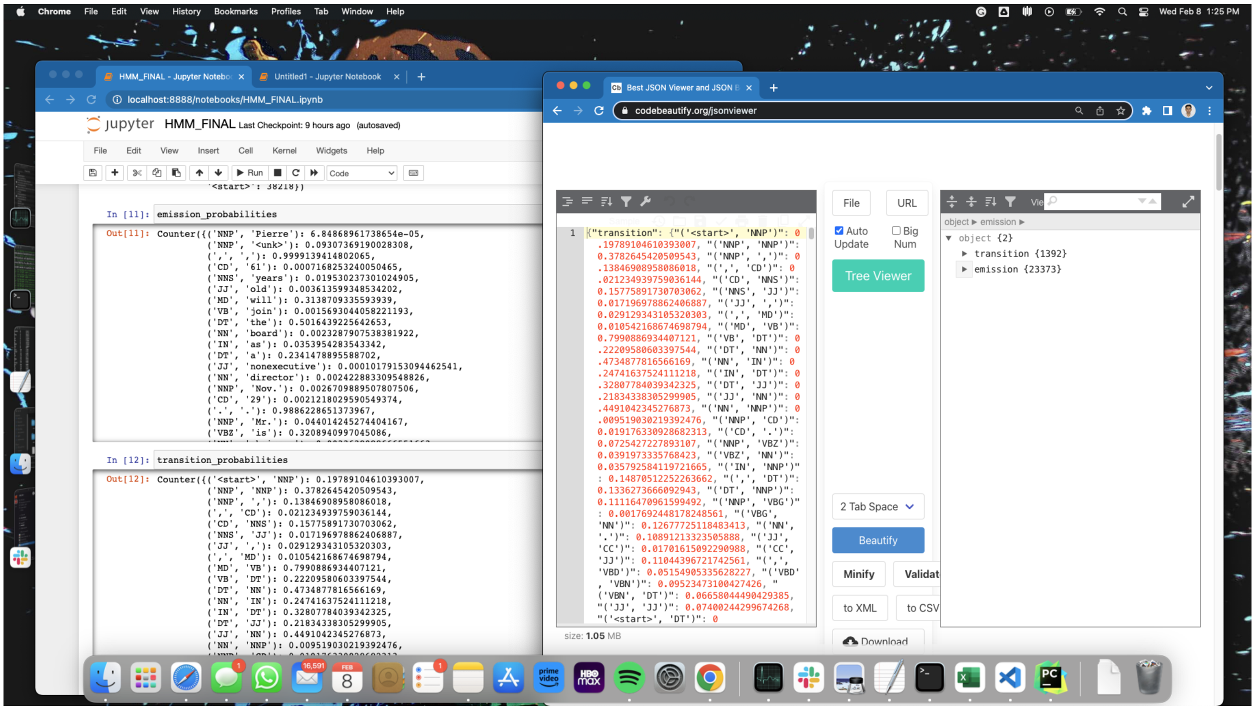
```python
# Task 2: Model Learning (20 points)
def model_learning(word_counter_final, data):
    #Get the emission and transition count first
    #Maintain a tag_counter that counts the number of occurrences of each tag
    tag_counter = Counter()
    emission_probabilities, transition_probabilities = Counter(), Counter()
    for i, sentence in enumerate(data):
        for j, word_desc in enumerate(sentence):
            #If word is not in word_counter_final, we handle it as <unk> special token.
            if word_desc[1] not in word_counter_final:
                word_desc[1] = '<unk>'
            tag_counter[word_desc[2]] += 1
            emission_probabilities[(word_desc[2], word_desc[1])] += 1
            #For the first word, use <start> as the previous token.
            if j == 0:
                transition_probabilities[('<start>', word_desc[2])] += 1
            #For the last word, we don't have to calculate transition_probabilities.
            elif j == len(sentence):
                continue
            else:
                transition_probabilities[(sentence[j - 1][2], sentence[j][2])] += 1

    tag_counter['<start>'] = len(data)

    #Use the tag_counter to normalize emission_probabilities and transition_probabilities
    for key, val in emission_probabilities.items():
        emission_probabilities[key] = val / tag_counter[key[0]]
    for key, val in transition_probabilities.items():
        transition_probabilities[key] = val / tag_counter[key[0]]

    # Write transition and emission probabilities to hmm.json
    js = {}
    t = {}
    e = {}
    for k, v in transition_probabilities.items():
        t[repr(k)] = v
    for k, v in emission_probabilities.items():
        e[repr(k)] = v
    js['transition'] = t
    js['emission'] = e
    with open("hmm.json", "w") as outfile:
        json.dump(js, outfile)

    return tag_counter, emission_probabilities, transition_probabilities
```



Q1) How many transition and emission parameters in your HMM?

- Emission parameters count: 1392
- Transition parameters count: 23372

Task 3: Greedy Decoding with HMM (30 points)

Steps:

1. Keep track of predicted tags for entire document in predicted_tags_greedy
2. Keep track of predicted tags for one sentence in word_tags
3. In the beginning start with a previous_tag of <start>. Required to calculate transition_probabilities
4. If word is not in word_counter_final, we handle it as <unk> special token
5. For each word, we try all possible tags and calculate the ep*tp
6. We assign the tag that gives maximum ep*tp to the word and proceed

```
def accuracy(ground_truth_tags, predicted_tags):
    return sum(
        [a == b for a, b in zip(ground_truth_tags, [word for sublist in predicted_tags for word in sublist])]) / len(
        ground_truth_tags)
```

```
# Task 3: Greedy Decoding with HMM (30 points)
def greedy_decoding(data, word_counter_final, tag_counter, emission_probabilities, transition_probabilities):
    #Keep track of predicted tags for entire document in predicted_tags_greedy
    predicted_tags_greedy = []
    unique_tags = [x for x in tag_counter.keys() if x != "<start>"]
    for sentence in deepcopy(data):
        #Keep track of predicted tags for one sentence in word_tags
        word_tags = []
        #In the beginning start with a previous_tag of <start>. Required to calculate transition_probabilities
        prev_tag = '<start>'
        for word_desc in sentence:
            s = -1
            #If word is not in word_counter_final, we handle it as <unk> special token
            if word_desc[1] not in word_counter_final:
                word_desc[1] = '<unk>'
            #For each word, we try all possible tags and calculate the ep*tp
            #We assign the tag that gives maximum ep*tp to the word and proceed
            for tag in unique_tags:
                ep = emission_probabilities.get((tag, word_desc[1]), 0)
                tp = transition_probabilities.get((prev_tag, tag), 0)

                if (ep * tp) > s:
                    s = ep * tp
                    temp_tag = tag

            prev_tag = temp_tag
            word_tags.append(temp_tag)
#           print(word_tags)
        predicted_tags_greedy.append(word_tags)
    return predicted_tags_greedy
```

```
dev_greedy_tags = greedy_decoding(dev_data, word_counter_final, tag_counter, emission_probabilities,
                                  transition_probabilities)
print(f"Accuracy dev_data with Greedy Decoding = {accuracy(ground_truth_tags, dev_greedy_tags)}")
```

```
Accuracy dev_data with Greedy Decoding = 0.9298615748891992
```

Q1) What is the accuracy on the dev data?

- The accuracy is 92.98% using HMM with Greedy Decoding. Note that I will add a heuristic later.

Task 4: Viterbi Decoding with HMM (30 points)

Steps:

1. Keep track of predicted tags for the entire document in predicted_tags_greedy
2. For each sentence run viterbi_one_sentence which returns predicted tags for a sentence
3. Initialize viterbi_matrix which has viterbi values for possible tags for words of a sentence
4. Initialize backpointer_matrix which keeps track of tag that gives maximum viterbi values
5. Use the initial probability distribution to set the first column of viterbi_matrix
6. Handle unknown words as <unk>
7. The value of each cell of viterbi_matrix is computed by
8. recursively taking the most probable path that could lead us to this tag.
9. Store the index to this path in backpointer matrix
10. Find the most likely tag of the last word using the last column of viterbi_matrix
11. Follow the backpointer_matrix corresponding to this tag to decode the predicted tags

```python
#Task 4: Viterbi Decoding with HMM (30 Points)
def viterbi_decoding(data, word_counter_final, tag_counter, emission_count, transition_count):
    unique_tags = [x for x in tag_counter.keys() if x != "<start>"]

    def viterbi_one_sentence(sentence, unique_tags):
        n = len(sentence)
        sent = [elem[1] for elem in sentence]
        #Initialize viterbi_matrix which has viterbi values for possible tags for words of a sentence
        viterbi_matrix = np.zeros((len(unique_tags), len(sent)))
        #Initialize backpointer_matrix which keeps track of tag that gives maximum viterbi values
        backpointer_matrix = np.zeros((len(unique_tags), len(sent))).astype(int)
        for i, tag in enumerate(unique_tags):
            #Use the initial probability distribution to set the first column of viterbi_matrix
            first_word = sent[0] if sent[0] in word_counter_final else '<unk>'
            viterbi_matrix[i][0] = transition_count.get(('<start>', tag), 0) * emission_count.get((tag, first_word),0)
        for i, word in enumerate(sent[1:]):
            for j, tag in enumerate(unique_tags):
                #Handle unknown words as <unk>
                if word not in word_counter_final:
                    word = '<unk>'
                probs = [(viterbi_matrix[x][i] * transition_count.get((unique_tags[x], tag),
                                                                       0) * emission_count.get((tag, word), 0)) for x
                         in range(len(unique_tags))]

                #The value of each cell of viterbi_matrix is computed by
                #recursively taking the most probable path that could lead us to this tag.
                viterbi_matrix[j][i + 1] = max(probs)
                #Store the index to this path in backpointer matrix
                backpointer_matrix[j][i + 1] = np.argmax(np.array(probs))

        #Find the most likely tag of the last word using the last column of viterbi_matrix
        #Follow the backpointer_matrix corresponding to this tag to decode the predicted tags
        best_path_pointer = np.argmax(viterbi_matrix[:, n - 1])
        tags_for_the_sentence = []
        best_decoded_path = []
        for k in range(n - 1, -1, -1):
            best_decoded_path.append(best_path_pointer)
            best_path_pointer = backpointer_matrix[best_path_pointer][k]
        best_decoded_path = reversed(best_decoded_path)
        for tag_ind in best_decoded_path:
            tags_for_the_sentence.append(unique_tags[tag_ind])
        return tags_for_the_sentence

    #Keep track of predicted tags for the entire document in predicted_tags_greedy
    predicted_tags_viterbi = []
    for sentence in data:
        #For each sentence run viterbi_one_sentence which returns predicted tags for a sentence
        predicted_tags_viterbi.append(viterbi_one_sentence(sentence, unique_tags))

    return predicted_tags_viterbi
```

```python
dev_viterbi_tags = viterbi_decoding(dev_data, word_counter_final, tag_counter, emission_probabilities,
                                    transition_probabilities)
print(f"Accuracy dev_data with Viterbi Decoding = {accuracy(ground_truth_tags, dev_viterbi_tags)}")
```

```
Accuracy dev_data with Viterbi Decoding = 0.9436813186813187
```

Q1) What is the accuracy on the dev data?

- The accuracy on the dev data is 94.36% using HMM with Viterbi Decoding. Please check the heuristic below.

HEURISTIC:

Idea:

1. While going through the training data, looks for words whose tags are always the same. For example: comma(,) always has tag comma(,).
2. This will help us in tagging all the punctuations correctly. In addition, it will tag some proper nouns, numbers and conjunctions correctly.
3. Since our training data unambiguously gave us these tags, our best prediction should be these tags.

```python
def heuristic_learn(train_data):
    unambiguous_tags = dict()
    for i, sent in enumerate(train_data):
        for j, word in enumerate(sent):
            if word[1] in unambiguous_tags and unambiguous_tags[word[1]] != word[2]:
                unambiguous_tags[word[1]] = '<ambiguous>'
                continue
            unambiguous_tags[word[1]] = word[2]
    return unambiguous_tags

def heuristic_apply(dev_data, unambiguous_tags, predicted_tags):
    corrected_count = 0
    for i, sent in enumerate(dev_data):
        for j, word in enumerate(sent):
            if word[1] in unambiguous_tags and unambiguous_tags[word[1]] != '<ambiguous>':
                if unambiguous_tags[word[1]] != predicted_tags[i][j]:
                    corrected_count += 1
                    print(f"For word {word} HMM predicted tag: {predicted_tags[i][j]}, Heuristic tag: {unambiguous_tags
                    predicted_tags[i][j] = unambiguous_tags[word[1]]

    return corrected_count
```

```python
# Print With heuristic
unambiguous_tags = heuristic_learn(train_data)

greedy_corrected_count = heuristic_apply(dev_data, unambiguous_tags, dev_greedy_tags)
print(f"\nAccuracy dev_data with Greedy Decoding and Heuristic = {accuracy(ground_truth_tags, dev_greedy_tags)}")
print(f"Heuristic corrected {greedy_corrected_count} in Greedy output.\n")

viterbi_corrected_count = heuristic_apply(dev_data, unambiguous_tags, dev_viterbi_tags)
print(f"\nAccuracy dev_data with Viterbi Decoding and Heuristic = {accuracy(ground_truth_tags, dev_viterbi_tags)}")
print(f"Heuristic corrected {viterbi_corrected_count} in Heuristic output.")
```

For word ['7', '--', ':'] HMM predicted tag: NNP, Heuristic tag: :
For word ['24', 'their', 'PRP$'] HMM predicted tag: NNP, Heuristic tag: PRP$
For word ['5', 'camera', 'NN'] HMM predicted tag: NNP, Heuristic tag: NN
For word ['6', '...', ':'] HMM predicted tag: NNP, Heuristic tag: :
For word ['16', '$', '$'] HMM predicted tag: NNP, Heuristic tag: $
For word ['42', '``', '``'] HMM predicted tag: NNP, Heuristic tag: ``
For word ['11', ',', ','] HMM predicted tag: NNP, Heuristic tag: ,
For word ['13', ')', '-RRB-'] HMM predicted tag: NNP, Heuristic tag: -RRB-
For word ['13', 'customer', 'NN'] HMM predicted tag: NNP, Heuristic tag: NN
For word ['22', 'again', 'RB'] HMM predicted tag: NNP, Heuristic tag: RB
For word ['5', 'or', 'CC'] HMM predicted tag: NNP, Heuristic tag: CC
For word ['16', 'attention', 'NN'] HMM predicted tag: NNP, Heuristic tag: NN
For word ['7', '.', '.'] HMM predicted tag: NNP, Heuristic tag: .
For word ['31', '.', '.'] HMM predicted tag: NNP, Heuristic tag: .
For word ['17', '--', ':'] HMM predicted tag: NNP, Heuristic tag: :
For word ['6', 'gone', 'VBN'] HMM predicted tag: NNP, Heuristic tag: VBN

For word ['16', 'whose', 'WP$'] HMM predicted tag: NNP, Heuristic tag: WP$
For word ['8', 'today', 'NN'] HMM predicted tag: NNP, Heuristic tag: NN

Accuracy dev_data with Greedy Decoding and Heuristic = 0.9299981786169631
Heuristic corrected 18 in Greedy output.

For word ['1', '``', '``'] HMM predicted tag: NNP, Heuristic tag: ``
For word ['2', '`', '``'] HMM predicted tag: NNP, Heuristic tag: ``
For word ['5', 'your', 'PRP$'] HMM predicted tag: NNP, Heuristic tag: PRP$
For word ['6', '...', ':'] HMM predicted tag: NNP, Heuristic tag: :
For word ['7', ',', ','] HMM predicted tag: NNP, Heuristic tag: ,
For word ['9', 'he', 'PRP'] HMM predicted tag: NNP, Heuristic tag: PRP
For word ['10', 'would', 'MD'] HMM predicted tag: NNP, Heuristic tag: MD
For word ['12', ',', ','] HMM predicted tag: NNP, Heuristic tag: ,
For word ['13', "''", "''"] HMM predicted tag: NNP, Heuristic tag: ''
For word ['16', 'authors', 'NNS'] HMM predicted tag: NNP, Heuristic tag: NNS
For word ['17', '.', '.'] HMM predicted tag: NNP, Heuristic tag: .
For word ['10', 'problems', 'NNS'] HMM predicted tag: NNP, Heuristic tag: NNS
For word ['13', 'year', 'NN'] HMM predicted tag: NNP, Heuristic tag: NN
For word ['14', ',', ','] HMM predicted tag: NNP, Heuristic tag: ,
For word ['15', 'when', 'WRB'] HMM predicted tag: NNP, Heuristic tag: WRB
For word ['16', '$', '$'] HMM predicted tag: NNP, Heuristic tag: $
For word ['17', '65', 'CD'] HMM predicted tag: NNP, Heuristic tag: CD
For word ['18', 'million', 'CD'] HMM predicted tag: NNP, Heuristic tag: CD
For word ['23', 'businessman', 'NN'] HMM predicted tag: NNP, Heuristic tag: NN
For word ['26', 'went', 'VBD'] HMM predicted tag: NNP, Heuristic tag: VBD
For word ['28', '.', '.'] HMM predicted tag: NNP, Heuristic tag: .
For word ['40', ',', ','] HMM predicted tag: NNP, Heuristic tag: ,
For word ['41', 'whose', 'WP$'] HMM predicted tag: NNP, Heuristic tag: WP$
For word ['42', '``', '``'] HMM predicted tag: NNP, Heuristic tag: ``
For word ['44', 'girl', 'NN'] HMM predicted tag: NNP, Heuristic tag: NN
For word ['45', 'now', 'RB'] HMM predicted tag: NNP, Heuristic tag: RB
For word ['48', 'every', 'DT'] HMM predicted tag: NNP, Heuristic tag: DT
For word ['50', 'she', 'PRP'] HMM predicted tag: NNP, Heuristic tag: PRP
For word ['51', 'sees', 'VBZ'] HMM predicted tag: NNP, Heuristic tag: VBZ
For word ['53', 'newspaper', 'NN'] HMM predicted tag: NNP, Heuristic tag: NN
For word ['54', '.', '.'] HMM predicted tag: NNP, Heuristic tag: .
For word ['55', "''", "''"] HMM predicted tag: NNP, Heuristic tag: ''
For word ['11', ',', ','] HMM predicted tag: NNP, Heuristic tag: ,
For word ['12', 'how', 'WRB'] HMM predicted tag: NNP, Heuristic tag: WRB
For word ['13', ')', '-RRB-'] HMM predicted tag: NNP, Heuristic tag: -RRB-
For word ['14', '?', '.'] HMM predicted tag: NNP, Heuristic tag: .

Accuracy dev_data with Viterbi Decoding and Heuristic = 0.9439545261368466
Heuristic corrected 36 in Heuristic output.

PREDICTING ON TEST DATA

```
with open('data/test') as file:
    s = file.read().splitlines()
    test_data = []
    temp = []
    for d in s:
        if d != '':
            word_desc = d.split('\t')
            temp.append(word_desc)
        else:
            test_data.append(temp)
            temp = []
    test_data.append(temp)
# print(test_data)
test_greedy_tags = greedy_decoding(test_data, word_counter_final, tag_counter, emission_probabilities,
                                   transition_probabilities)

heuristic_apply(test_data, unambiguous_tags, test_greedy_tags)

with open('greedy.out', 'w') as file:
    for i, sentence in enumerate(test_data):
        for j, word_desc in enumerate(sentence):
            write = f"{j + 1}\t{word_desc[1]}\t{test_greedy_tags[i][j]}\n"
            file.write(write)
        file.write('\n')

test_viterbi_tags = viterbi_decoding(test_data, word_counter_final, tag_counter, emission_probabilities,
                                     transition_probabilities)

heuristic_apply(test_data, unambiguous_tags, test_viterbi_tags)
#
with open('viterbi.out', 'w') as file:
    for i, sentence in enumerate(test_data):
        for j, word_desc in enumerate(sentence):
            write = f"{j + 1}\t{word_desc[1]}\t{test_viterbi_tags[i][j]}\n"
            file.write(write)
        file.write('\n')
```