CSCI 544 - Applied Natural language processing Assignment-3

Name: Anne Sai Venkata Naga Saketh

USC Email: annes@usc.edu

USC ID: 3725520208

In [1]:
```
# Importing the required libraries
import csv
# Library for Hidden Markov Models (HMM)
import hmmlearn
# Library for creating defaultdicts, a subclass of dict
from collections import defaultdict, Counter
# Library providing functions that map Python operators to corresponding
import operator
# Importing the JSON library to read and write the json files
import json
```

Task 1: Vocabulary Creation

In [2]:
```
# Read the training data
with open('./data/train', 'r') as file:
    train_data = file.readlines()

# Print first two sentences for demonstration
print("Train data:")
for sentence in train_data[:2]:
    print(sentence)

# Read the testing data
with open('./data/test', 'r') as file:
    test_data = file.readlines()

# Print first two sentences for demonstration
print("Test data:")
for sentence in test_data[:2]:
    print(sentence)

# Read the testing data
with open('./data/dev', 'r') as file:
    dev_data = file.readlines()

# Print first two sentences for demonstration
print("Dev data:")
for sentence in dev_data[:2]:
    print(sentence)
```

```
Train data:
1       Pierre  NNP

2       Vinken  NNP

Test data:
1       Influential

2       members

Dev data:
1       The     DT

2       Arizona NNP
```

In [3]:

```python
# Set the threshold for word frequency
frequency_threshold = 3

# Initialize dictionaries to store word and tag frequencies, and count se
w_f, t_f, s_c = defaultdict(int), defaultdict(int), 0

# Initialize a list to store file data with start tokens
f_d = ["<s>"]

# Iterate through each line in the training data
for line in train_data:

    # Check if the line is empty, indicating end of a sentence
    if line.strip() == "":

        # Add start token to file data
        f_d.append("<s>")

        # Increment sentence count
        s_c += 1
    else:
        speech = line.strip().split("\t")

        # Check if the line has three parts (word, tag, frequency)
        # Skip this line if it doesn't have three parts
        if len(speech) != 3:
            continue

        # Extract word and tag
        w, t = speech[1], speech[2]

        # Increment word frequency
        w_f[w] += 1

        # Increment tag frequency
        t_f[t] += 1

        # Add line to file data
        f_d.append(line.strip())

# Filter vocabulary based on frequency threshold
```

```
vocabulary = {w: f for w, f in w_f.items() if f >= frequency_threshold}

# Add <unk> token for infrequent words
vocabulary["<unk>"] = sum(f for w, f in w_f.items() if f < frequency_thre

# Write vocabulary to a file
with open("vocab.txt", "w") as vocabulary_file:
    # Write <unk> token with frequency
    vocabulary_file.write("<unk>\t0\t{}\n".format(vocabulary["<unk>"]))

    # Write each word with its index and frequency to the file, sorted by
    sorted_vocabulary = sorted(vocabulary.items(), key=lambda x: x[1], re
    for idx, (w, f) in enumerate(sorted_vocabulary, start=1):
        vocabulary_file.write("{}\t{}\t{}\n".format(w, idx, f))

# Print vocabulary size after replacement and total occurrences of '<unk>
print(f"The Vocabulary size after replacement of the least occurring occu
print(f"Total occurrences of ‘<unk>’ token are: {vocabulary['<unk>']}")
```

```
The Vocabulary size after replacement of the least occurring occurrences
by the threshold: 16920
Total occurrences of ‘<unk>’ token are: 32537
```

Task 2: Model Learning

In [4]:

```
# Initialize dictionaries to store emission and transition probabilities
e_probabilities, t_probabilities = defaultdict(int), defaultdict(int)

# Initialize previous tag variable with start token
prev_tag = "<s>"

# Iterate through each line in the file data
for line in f_d:

    # Check if the line is a start token
    if line == "<s>":
        prev_tag = "<s>"  # Reset previous tag to start token
        continue

    # Split the line by tab
    speech = line.split("\t")
    w, cur_tag = speech[1], speech[2]

    # Replace infrequent words with <unk>
    w = w if w in vocabulary else "<unk>"

    # Increment emission probability
    e_probabilities[(cur_tag, w)] += 1

    # Update transition probabilities based on previous and current tags
    if prev_tag != "<s>":
        t_probabilities[(prev_tag, cur_tag)] += 1
    else:

        # Increment start transition probability
        t_probabilities[("start", cur_tag)] += 1
```

```
    prev_tag = cur_tag

# Normalize emission probabilities by tag frequency
for key in e_probabilities:
    e_probabilities[key] /= t_f[key[0]]

# Normalize transition probabilities by tag frequency or sentence count
for key in t_probabilities:
    if key[0] == "start":
        t_probabilities[key] /= s_c
    else:
        t_probabilities[key] /= t_f[key[0]]

# Print number of transition and emission parameters
print(f"Number of transition parameters in the hmm.json file are: {len(e_
print(f"Number of emission parameters in the hmm.json file are: {len(t_pr

# Convert keys to string for emission and transition probabilities
e_k = {f"({tag},{word})": prob for (tag, word), prob in e_probabilities.i
t_k = {f"({prev_tag},{next_tag})": prob for (prev_tag, next_tag), prob in

# Create an HMM model dictionary
model = {"Transition": t_k, "Emission": e_k}

# Save HMM model to a JSON file
with open('hmm.json', 'w') as json_file:
    json.dump(model, json_file, indent=4)
```

```
Number of transition parameters in the hmm.json file are: 23373
Number of emission parameters in the hmm.json file are: 1392
```

Task 3: Greedy Decoding with HMM

In [5]:

```
def load_hmm_from_json(file_path):
    with open(file_path, 'r') as f:
        data = json.load(f)

    # Create a dictionary containing HMM parameters with keys "Transition
    return {"Transition": data["Transition"], "Emission": data["Emission"

# Path to the JSON file containing HMM parameters
hmm_file_path = 'hmm.json'

# Load HMM parameters from the JSON file
model = load_hmm_from_json(hmm_file_path)
```

Testing on the Dev Data

In [6]:

```
# Specify the output file name
output_file_name = "greedy_dev.out"

# Initialize a list to store data to be written
w_d = []
```

```
# Initialize previous tag variable with start token
prev_tag = "start"

# Open the development data file for reading and output file for writing
with open(output_file_name, "w") as output_file:

    # Iterate through each line in the development data
    for line in dev_data:
        w = line.split("\t")

        # Check if the line contains only one element, indicating start o
        if len(w) == 1:

            # Reset previous tag to start token
            prev_tag = "start"

            # Append line to write data
            w_d.append(line)
            continue
        else:
            # Extract index and current word
            idx, cur_word = w[0].strip(), w[1].strip()

            # Replace infrequent words with <unk>
            if cur_word not in vocabulary:
                cur_word = "<unk>"

            # Initialize probability and temporary tag variables
            prob_val, temp_tag = 0, ""

            # Iterate through each tag in the tag frequency dictionary
            for tag_iter in t_f:

                # Check emission and transition probabilities for the cur
                e_c = (tag_iter, cur_word)
                emission_prob_value = e_probabilities.get(e_c, 0)
                t_c = (prev_tag, tag_iter)
                transition_prob_value = t_probabilities.get(t_c, 0)
                current_prob_val = emission_prob_value * transition_prob_

                # Update probability and temporary tag if current probabi
                if current_prob_val >= prob_val:
                    prob_val, temp_tag = current_prob_val, tag_iter

            # Update previous tag with temporary tag
            prev_tag = temp_tag

            # Construct the line to be written
            cur_line = f"{idx}\t{cur_word}\t{prev_tag}\n"

            # Append line to write data
            w_d.append(cur_line)

    # Write all the data to the output file
    output_file.writelines(w_d)

# Printing the output
```

```
print("Output file: greedy_dev.out has been generated successfully")
```

```
Output file: greedy_dev.out has been generated successfully
```

In [7]:
```
!python eval.py -p greedy_dev.out -g ./data/dev
```

```
total: 131768, correct: 122390, accuracy: 92.88%
```

Testing on the Test Data

In [8]:
```
# Specify the output file name
output_file_name = "greedy.out"

# Initialize a list to store data to be written
w_d = []

# Initialize previous tag variable with start token
prev_tag = "start"

# Open the development data file for reading and output file for writing
with open(output_file_name, "w") as output_file:

    # Iterate through each line in the development data
    for line in test_data:
        w = line.split("\t")

        # Check if the line contains only one element, indicating start o
        if len(w) == 1:

            # Reset previous tag to start token
            prev_tag = "start"

            # Append line to write data
            w_d.append(line)
            continue
        else:
            # Extract index and current word
            idx, cur_word = w[0].strip(), w[1].strip()

            # Replace infrequent words with <unk>
            if cur_word not in vocabulary:
                cur_word = "<unk>"

            # Initialize probability and temporary tag variables
            prob_val, temp_tag = 0, ""

            # Iterate through each tag in the tag frequency dictionary
            for tag_iter in t_f:

                # Check emission and transition probabilities for the cur
                e_c = (tag_iter, cur_word)
                emission_prob_value = e_probabilities.get(e_c, 0)
                t_c = (prev_tag, tag_iter)
                transition_prob_value = t_probabilities.get(t_c, 0)
                current_prob_val = emission_prob_value * transition_prob_
```

```
                # Update probability and temporary tag if current probabi
                if current_prob_val >= prob_val:
                    prob_val, temp_tag = current_prob_val, tag_iter

            # Update previous tag with temporary tag
            prev_tag = temp_tag

            # Construct the line to be written
            cur_line = f"{idx}\t{cur_word}\t{prev_tag}\n"

            # Append line to write data
            w_d.append(cur_line)

    # Write all the data to the output file
    output_file.writelines(w_d)

# Printing the output
print("Output file: greedy.out has been generated successfully")
```

```
Output file: greedy.out has been generated successfully
```

Task 4: Viterbi Decoding with HMM

In [9]:
```
def viterbi(o_w, s_l, e_probabilities, t_probabilities):
    # Calculate the total number of observations and states
    tot_obs = len(o_w)
    tot_s = len(s_l)

    # Initialize the Viterbi matrix and backpointers matrix
    v_m = [[0 for _ in range(tot_s)] for _ in range(tot_obs)]
    backtracking = [[0 for _ in range(tot_s)] for _ in range(tot_obs)]

    # Initialization step
    for ind in range(tot_s):

        # Calculate transition and emission probabilities for the first o
        t_prob = t_probabilities.get(('start', s_l[ind]), 1e-10)
        e_prob = e_probabilities.get((s_l[ind], o_w[0]), 1e-10)

        # Initialize the first column of the Viterbi matrix and backpoint
        v_m[0][ind] = t_prob * e_prob
        backtracking[0][ind] = 0

    # Recursion step
    for t_s in range(1, tot_obs):
        for ind in range(tot_s):

            # Calculate the maximum probability and the corresponding pre
            m_p, optimal_state = max(
                (v_m[t_s-1][prev_state] * t_probabilities.get((s_l[prev_s
                for prev_state in range(tot_s))

            # Update the current cell in the Viterbi matrix and backpoint
            v_m[t_s][ind] = m_p
            backtracking[t_s][ind] = optimal_state

    # Termination step
    final_t_s = tot_obs - 1
    # Find the final state with the highest probability
    optimal_state = max(range(tot_s), key=lambda s: v_m[final_t_s][s])

    # Path backtracking
    minimal_route = [optimal_state]
    for t_s in range(tot_obs - 1, 0, -1):

        # Insert the best previous state at the beginning of the optimal
        minimal_route.insert(0, backtracking[t_s][minimal_route[0]])

    # Convert state indices to state labels and return the optimal path
    return [s_l[region] for region in minimal_route]


# Assuming emission_probs is a dictionary where keys are tuples (tag, wor
# Extract unique tags from the keys of emission_probs
states = set(tag for tag, _ in e_probabilities.keys())
```

In [10]:
```
# If 'states' was originally defined as a set, convert it to a list
states = list(states)

# Determine the output file based on the data type
output_name = "viterbi_dev.out"

viterbi_result = []

idx_data, cw_data, o_w = [], [], []

for line in dev_data:

    # Sentence boundary
    if len(line.strip()) == 0:

        if o_w:
            # Perform Viterbi decoding on observed words
            wots = viterbi(o_w, states, e_probabilities, t_probabilities)

            # Append decoded tags along with word indices and original wo
            viterbi_result.extend(f"{idx}\t{word}\t{tag}\n" for idx, word

        # Reset for next sentence
        o_w, idx_data, cw_data = [], [], []

        viterbi_result.append("\n")
        continue

    idx, word = line.strip().split("\t")[:2]
    idx_data.append(idx)
    cw_data.append(word)

    # Replace out-of-vocabulary words with <unk>
    o_w.append(word if word in vocabulary else "<unk>")

# Write Viterbi output to file
with open(output_name, "w") as viterbi_file:
    viterbi_file.writelines(viterbi_result)

print("The output has been written into the file viterbi_dev.out")
```

```
The output has been written into the file viterbi_dev.out
```

In [11]:
```
!python eval.py -p viterbi_dev.out -g ./data/dev
```

```
'1\tThat\tDT' '38\t.\t.' 131751
total: 131751, correct: 124384, accuracy: 94.41%
```

In [12]:
```
# If 'states' was originally defined as a set, convert it to a list
states = list(states)

# Determine the output file based on the data type
output_name = "viterbi.out"

viterbi_result = []

idx_data, cw_data, o_w = [], [], []

for line in test_data:

    # Sentence boundary
    if len(line.strip()) == 0:

        if o_w:
            # Perform Viterbi decoding on observed words
            wots = viterbi(o_w, states, e_probabilities, t_probabilities)

            # Append decoded tags along with word indices and original wo
            viterbi_result.extend(f"{idx}\t{word}\t{tag}\n" for idx, word

        # Reset for next sentence
        o_w, idx_data, cw_data = [], [], []

        viterbi_result.append("\n")
        continue

    idx, word = line.strip().split("\t")[:2]
    idx_data.append(idx)
    cw_data.append(word)

    # Replace out-of-vocabulary words with <unk>
    o_w.append(word if word in vocabulary else "<unk>")

# Write Viterbi output to file
with open(output_name, "w") as viterbi_file:
    viterbi_file.writelines(viterbi_result)

print("The output has been written into the file viterbi.out")
```

```
The output has been written into the file viterbi.out
```