CSCI 544: Assignment 2

This report encompasses the description of my implementation for Assignment 2 in CSCI 544. It consists of two main sections:

1. **Vocabulary Creation**:

As per the assignment instructions, the training, development, and testing data were already preprocessed and did not require further cleaning. Using python's *json* library, I loaded the train, test and dev sets. Each dataset is a list of dictionaries having the index, sentence (list of words) and labels already segregated, and no further processing is needed.

To fulfill the vocabulary creation task, I needed to select an appropriate threshold value for unknown word occurrences. To determine the ideal threshold, I conducted several test runs to evaluate vocabulary size and the number of unknown words for different thresholds. Based on the results, I decided on a threshold of **2**, striking a balance between manageable vocabulary size and a reasonable number of unknown words.

Here are the statistics :

- Threshold set: **2**
- Vocabulary Size: **16920**
- Unknown Word Occurrences: **32537**

The vocabulary is stored as a dictionary with key as word and it corresponding value as the number of occurrences in the training data. All the words present in the training data that are not a part of the vocabulary dictionary are treated as "unknown words" or "<unk>".
In an attempt to better handle the unknown words, I implemented the concept of pseudo-words, mentioned in the reading material. As per the reference paper, I tried 12 types of pseudo-words, the 13th one being "other" or the default, "<unk>". For every word in the training data, if it is nor in the vocabulary dictionary, it is assigned its respective pseudo-word via a self-defined function.

2. **Model Learning**:

I identified that there are 45 distinct tags in the training data using a set and updating this set with every new tag. This list of 45 tags was used to construct the transition dictionary.

To simplify the creation of the transmission and emission dictionaries, I designed several functions that calculated state transition (count(S->S')) and state emission (count(S->X)) occurrences. These helper functions streamlined the process.

For the transition dictionary, I dedicated a separate loop to define the beginning of a sentence, assigning it a '<start>' tag. I calculated transition probabilities from this starting tag to all other tags, saving them in the transition dictionary.

```
No. of transition parameters: 2070
No. of emission parameters: 761492
```

3. **Greedy Decoding**:

After preparing the emission and transition dictionaries, I implemented the greedy decoding algorithm. This algorithm processes one state at a time, utilizing a function designed to handle a single state and return its most probable tag. Supporting functions were also created to access probabilities.

The accuracy of the Greedy Decoding algorithm, when tested on the dev set, achieved 93.94%.

4. **Viterbi Decoding**:

The final task involved implementing Viterbi Decoding on the same dataset. Initially, I attempted to implement the algorithm using nested loops, which resulted in extended computation times. To address this, I adopted a matrix multiplication approach, which requires transition, emission, and initial probability matrices. Functions were developed to generate these matrices, enabling the implementation of Viterbi Decoding.

The Viterbi Decoding method creates a matrix V (for storing computed values) and a 1-dimensional array for backtracking. During the loop, the most probable state in a given sequence is recorded in the backtracking array, which is then reversed to determine the correct order of predicted states.

After implementing the matrix version (dynamic programming version) of Viterbi Decoding, based on the advice given on Piazza, I tried using log values of emission and transmission, adding those log-probabilities instead of directly multiplying the probabilities to prevent underflow (i.e. a really small probability chain resulting in value 0 due to computational bounds)

To further improve upon accuracy, I tried a Laplace smoothing on the emission and transition matrices with values as small as 1e-8.

The accuracy of Viterbi Decoding, when tested on the dev data, reached 94.98%.

The following pages, which has the notebook cells, is my implementation of the homework.

In [1]:
```
import json
import numpy as np
import pandas as pd
from collections import defaultdict
import json
from tqdm import tqdm
```

In [2]:
```
with open('train.json') as f:
    train_data = json.load(f)
```

Task 1: Vocabulary Creation

In [3]:
```
def create_vocabulary(data,threshold):
    vocab_dict = {}
    vocab_dict['<unk>']=0 #keeping 0 unknowns
    for point in data:
        for i in point['sentence']:
            if i not in vocab_dict:
                vocab_dict[i] = 1
            else:
                vocab_dict[i] +=1
    pop_list = []
    for i in vocab_dict:
        if i != '<unk>' and vocab_dict[i]<=threshold:
            vocab_dict['<unk>']+= vocab_dict[i]
            pop_list.append(i)
    for i in pop_list:
        vocab_dict.pop(i)
    vocab_dict= dict(sorted(vocab_dict.items(), key=lambda x:x[1],reverse =
    print(f"Threshold Set: {threshold}")
    print(f"Vocabulary Length: {len(vocab_dict)}")
    print(f"Count of Unknown Occurrences: {vocab_dict['<unk>']}")
    return vocab_dict
```

In [4]:
```
print("Trying out multiple thresholds of vocabulary: ")
for i in range(1,11):
    vocab = create_vocabulary(train_data,i)
    print("_"*40)
```

```
Trying out multiple thresholds of vocabulary:
Threshold Set: 1
Vocabulary Length: 23183
Count of Unknown Occurrences: 20011
________________________________________
Threshold Set: 2
Vocabulary Length: 16920
Count of Unknown Occurrences: 32537
________________________________________
Threshold Set: 3
Vocabulary Length: 13751
Count of Unknown Occurrences: 42044
________________________________________
Threshold Set: 4
Vocabulary Length: 11688
Count of Unknown Occurrences: 50296
________________________________________
Threshold Set: 5
Vocabulary Length: 10236
Count of Unknown Occurrences: 57556
________________________________________
Threshold Set: 6
Vocabulary Length: 9156
Count of Unknown Occurrences: 64036
________________________________________
Threshold Set: 7
Vocabulary Length: 8363
Count of Unknown Occurrences: 69587
________________________________________
Threshold Set: 8
Vocabulary Length: 7695
Count of Unknown Occurrences: 74931
________________________________________
Threshold Set: 9
Vocabulary Length: 7097
Count of Unknown Occurrences: 80313
________________________________________
Threshold Set: 10
Vocabulary Length: 6588
Count of Unknown Occurrences: 85403
________________________________________
```

In [5]:
```
# choosing threshold as 2
vocab = create_vocabulary(train_data,2)
```

```
Threshold Set: 2
Vocabulary Length: 16920
Count of Unknown Occurrences: 32537
```

In [6]:
```
sorted_vocab = sorted(vocab.items(), key=lambda x: (-x[1], x[0]))
sorted_vocab.remove(('<unk>', vocab['<unk>']))
sorted_vocab.insert(0, ('<unk>', vocab['<unk>']))

# Write to the file
with open('vocab.txt', 'w') as f:
    for idx, (word, freq) in enumerate(sorted_vocab):
        f.write(f"{word}\t{idx}\t{freq}\n")
```

In [7]:
```
def pseudo_word(word):
    """
    Convert a word into a pseudo-word based on its features.
    This method is based on the table from Bikel et al. (1999)
    with additional categories.
    """
    # Based on the table
    if len(word) == 2 and word.isdigit():
        return "<twoDigitNum>"

    if len(word) == 4 and word.isdigit():
        return "<fourDigitNum>"

    if any(char.isdigit() for char in word) and any(char.isalpha() for char
        return "<containsDigitAndAlpha>"

    if "-" in word and any(char.isdigit() for char in word):
        return "<containsDigitAndDash>"

    if "/" in word and any(char.isdigit() for char in word):
        return "<containsDigitAndSlash>"

    if "," in word and any(char.isdigit() for char in word):
        return "<containsDigitAndComma>"

    if "." in word and any(char.isdigit() for char in word):
        return "<containsDigitAndPeriod>"

    if word.isdigit() and not any([char in word for char in ["-", "/", ".",
        return "<othernum>"

    if word.isupper():
        return "<allCaps>"

    if len(word) == 2 and word[1] == "." and word[0].isupper():
        return "<capPeriod>"

    if word[0].isupper():
        return "<initCap>"

    if word.islower():
        return "<lowercase>"
    return "<unk>"
```

In [8]:
```
# removing unknown words from training data
for data_point in train_data:
    for i in range(len(data_point['sentence'])):
        if data_point['sentence'][i] not in vocab:
            data_point['sentence'][i] = pseudo_word(data_point['sentence'][
```

Task 2: Model Learning

In [9]:
```
def count_tag(state,data):
    """
    returns number of occurences of a POS tag in data
    """
    count = 0
    for data_point in data:
        count += data_point['labels'].count(state)
    return count

def count_transition(S,S1,data):
    """
    returns number of occurences of S->S1 in the training data
    """
    count = 0
    for data_point in data:
        labels = data_point['labels']
        for i in range(len(labels) - 1):
            if labels[i] == S and labels[i + 1] == S1:
                count += 1
    return count

def count_emmission(S,X,data):
    """
    returns number of occurences where state S emits word X in training dat
    """
    count = 0
    for data_point in data:
        for w, t in zip(data_point['sentence'], data_point['labels']):
            if w == X and t == S:
                count += 1
    return count

def count_tag_initial(S,data):
    count = 0
    for data_point in data:
        if data_point['labels'][0]==S:
            count += 1
    return count
```

In [10]:
```
unique_tags = set()  # Create an empty set to store unique tags
for item in train_data:
    unique_tags.update(item['labels'])
```

In [11]:
```
# computing transition probabilities
transition_probabilities = {}
words = list(unique_tags)
for i in tqdm(words, desc="Processing tags", ncols=100): #to track progress
    for j in words:
        name = i +","+j
        transition_probabilities[name] = count_transition(i,j,train_data)/c
```

```
Processing tags: 100%|████████████████████████████████████████████████| 45/
45 [02:16<00:00,  3.04s/it]
```

In [12]:
```
# adding initial probabilities to transition dictionary
for state in unique_tags:
    transition_probabilities[f"<start>,{state}"] = count_tag_initial(state,
```

In [13]:
```
# saving transition dictionary
with open("transition.json", "w") as json_file:
    json.dump(transition_probabilities, json_file, indent=4)
```

In [14]:
```python
# computing emission probabilties
def compute_emission_probabilities(train_data, vocab):
    # Default Dictionary to store word-label pair counts
    word_label_count = defaultdict(lambda: defaultdict(int))
    # Default Dictionary to store label counts
    label_count = defaultdict(int)
    # Set to store all unique labels
    all_labels = set()
    # Count occurrences for each word-label pair and each label
    for data_point in tqdm(train_data, desc="Processing data"):
        words = data_point['sentence']
        labels = data_point['labels']
        all_labels.update(labels)
        for word, label in zip(words, labels):
            word_label_count[word][label] += 1
            label_count[label] += 1

    # Calculate emission probabilities
    emission_probabilities = {}
    for word, labels in tqdm(word_label_count.items(), desc="Calculating pr
        for label, count in labels.items():
            key = f"{word},{label}"
            emission_probabilities[key] = count / label_count[label]

    # Add unseen word-label combinations with probability of 0
    for word in vocab.keys():
        for label in all_labels:
            key = f"{word},{label}"
            if key not in emission_probabilities:
                emission_probabilities[key] = 1e-12 #1e-8 #0.0 (worked the

    return emission_probabilities

# Compute and save the emission probabilities
emission_probabilities = compute_emission_probabilities(train_data, vocab)

with open("emission.json", "w") as json_file:
    json.dump(emission_probabilities, json_file, indent=4)
```

```
Processing data: 100%|██████████| 38218/38218 [00:00<00:00, 140035.
23it/s]
Calculating probabilities: 100%|██████████| 16930/16930 [00:00<00:00, 1070372.
27it/s]
```

In [15]:
```python
print(f"No. of transition parameters: {len(transition_probabilities)}")
print(f"No. of emission parameters: {len(emission_probabilities)}")
```

```
No. of transition parameters: 2070
No. of emission parameters: 761492
```

In [16]:
```
# helper functions for greedy algorithm
def get_transition_probability(S1, S2, transition_dict):
    key = f"{S1},{S2}"
    return transition_dict.get(key, 1e-5) # 1e-8 in case nothing was found

def get_emission_probability(S, X, emission_dict):
    key = f"{X},{S}"
    return emission_dict.get(key, emission_dict.get(f"<unk>,{S}",0.0))
```

In [17]:
```
hmm = {
    "transition":transition_probabilities,
    "emission":emission_probabilities
}
with open("hmm.json", "w") as json_file:
    json.dump(hmm, json_file, indent=4)
```

Task 3: Greedy Decoding with HMM

In [18]:
```
# greedy decoding
def greedy_decoding_state(S, word, transition_dict, emission_dict, state_se
    """
    implements the greedy decoding algorithm for a given state (1 step)
    """
    max_p = float('-inf') # lowest number possible
    selected_state = None

    for S1 in state_set:
        t = get_transition_probability(S, S1, transition_dict)
        e = get_emission_probability(S1, word, emission_dict)
        if t * e > max_p:
            max_p = t * e
            selected_state = S1
    return selected_state

def greedy_decoding(inputs, transition_dict, emission_dict, state_set):
    """
    implements the greedy decoding algorithm using greedy_decoding_state fo
    """
    state_sequence = []
    S = '<start>'  # Starting state
    for word in inputs:
        S = greedy_decoding_state(S, word, transition_dict, emission_dict,
        state_sequence.append(S)
    return state_sequence
```

In [20]:
```
#loading dev file
with open('dev.json') as f:
    dev_data = json.load(f)
```

In [21]:
```
for data_point in dev_data:
    for i in range(len(data_point['sentence'])):
        if data_point['sentence'][i] not in vocab:
            data_point['sentence'][i] = pseudo_word(data_point['sentence'][
```

In [22]:
```
def make_sentences(data):
    """
    takes data as input and returns a set of sentences and their correspond
    """
    return [[i['sentence'],i['labels']] for i in data]
```

In [23]:
```
dev_sentence_set = make_sentences(dev_data)
```

In [24]:
```
def get_accuracy(y_true, y_pred):
    """
    function that calculates accuracy of y_pred given y_true
    """
    if len(y_true) != len(y_pred):
        raise ValueError("Input lists must have the same length.")

    correct_predictions = sum([true == pred for true, pred in zip(y_true, y
    return correct_predictions / len(y_true)
```

In [25]:
```
def test_greedy(sentences,transition_dict,emmission_dict,set_states):
    """
    implements the greedy algorithm on all the sentences and calculates the
    """
    labels = []
    preds_greedy = []
    for sentence in sentences:
        preds_greedy += greedy_decoding(sentence[0],transition_dict,emmissi
        labels += sentence[1]
    return get_accuracy(labels,preds_greedy)
```

In [26]:
```
greedy_accuracy_dev = test_greedy(dev_sentence_set,transition_probabilities
print(f"Accuracy of Greedy Decoding on dev set: {greedy_accuracy_dev*100}%"
```

```
Accuracy of Greedy Decoding on dev set: 93.93631230647806%
```

Task 4: Viterbi Decoding with HMM

In [27]:
```
def generate_transition_matrix(transition_dict, ss):
    """
    Generates a transition matrix from transition_dict.
    Helps with faster calculation of the viterbi algorithm.
    """
    transition_matrix = [[transition_dict[ss[i] + ',' + ss[j]] for j in ran
    return np.array(transition_matrix)
```

In [28]:
```
def generate_emmission_matrix(emission_dict, vocab, ss):
    """
    Generates an emission matrix from an emission_dict.
    Helps with faster calculation of the viterbi algorithm.
    """
    emmission_matrix = [[get_emission_probability(ss[i],vocab[j],emission_d
    # storing the results as a dataframe (easier access of data)
    matrix = pd.DataFrame(np.array(emmission_matrix).T, index=vocab)
    return matrix
```

In [29]:
```
def generate_initial_probabilities(transition_dict, ss):
    """
    Generates initial probabilities based on transition_dict.
    """
    initial = [transition_dict.get('<start>,' + state, 1e-6) for state in s
    return initial
```

In [30]:
```
# generating emission matrix, transition matrix and initial probabilities
extra_vocab_list = ["<twoDigitNum>","<fourDigitNum>","<containsDigitAndAlph
                    "<containsDigitAndDash>","<containsDigitAndSlash>","<co
                     "<containsDigitAndPeriod>","<othernum>","<allCaps>","<
EM = generate_emmission_matrix(emission_probabilities,list(vocab.keys())+ex


TM = generate_transition_matrix(transition_probabilities,list(unique_tags))
initial = generate_initial_probabilities(transition_probabilities,list(uniq
```

In [31]:
```
def viterbi_decoding(inputs, transition_matrix, emission_matrix, initial_pr
    """
    Implements the Viterbi algorithm on the given inputs.

    inputs: a list of words of a given sentence
    transition_matrix: transition matrix
    emission_matrix: emission matrix
    initial_probabilities: initial probabilities
    state_set: set of possible states
    """
    N = len(inputs)
    n_states = len(initial_probabilities)

    V = np.zeros((N, n_states))

    # Initialize the first column of V based on initial probabilities and e
    first_word = inputs[0]
    if first_word in emission_matrix.index:
        V[0] = np.array(initial_probabilities) * emission_matrix.loc[first_
    else:
        V[0] = initial_probabilities * emission_matrix.loc['<unk>'].values

    # Fill in the rest of the Viterbi matrix
    for t in range(1, N):
        for s in range(n_states):
            word = inputs[t]
            if word in emission_matrix.index:
                V[t][s] = np.max(V[t-1] * transition_matrix[:, s]) * emissi
            else:
                V[t][s] = np.max(V[t-1] * transition_matrix[:, s]) * emissi

    # Backtracking to find the most likely sequence
    back_tracking = [np.argmax(V[-1])]
    for i in range(N-2, -1, -1):
        back_tracking.append(np.argmax(V[i] * transition_matrix[:, back_tra

    # Reverse the backtracking result to get the final sequence
    result = back_tracking[::-1]
    decoded_sequence = [state_set[j] for j in result]

    return decoded_sequence
```

In [32]:
```
EPSILON = 1e-10 # small constant used to avoid log(0) in calculations

def safe_log(x):
    """Compute the logarithm, but replace zeros or near-zeros with a small
    return np.log(np.where(np.abs(x) < EPSILON, EPSILON, x))

def viterbi_decoding_log(inputs, transition_matrix, emission_matrix, initia
    """
    Implements the Viterbi algorithm on the given inputs using log-space co
    """
    N = len(inputs)
    n_states = len(initial_probabilities)

    V = np.zeros((N, n_states)) - np.inf  # initialization of Viterbi matri

    # Initialize the first column of V based on initial probabilities and e
    first_word = inputs[0]
    if first_word in emission_matrix.index:
        V[0] = safe_log(initial_probabilities) + safe_log(emission_matrix.l
    else:
        V[0] = safe_log(initial_probabilities) + safe_log(emission_matrix.l

    # Fill in the rest of the Viterbi matrix
    for t in range(1, N):
        for s in range(n_states):
            word = inputs[t]
            if word in emission_matrix.index:
                V[t, s] = np.max(V[t-1] + safe_log(transition_matrix[:, s])
            else:
                V[t, s] = np.max(V[t-1] + safe_log(transition_matrix[:, s])

    # Backtracking to find the most likely sequence
    back_tracking = [np.argmax(V[-1])]
    for i in range(N-2, -1, -1):
        back_tracking.append(np.argmax(V[i] + safe_log(transition_matrix[:,

    # Reverse the backtracking result to get the final sequence
    result = back_tracking[::-1]
    decoded_sequence = [state_set[j] for j in result]

    return decoded_sequence
```

In [33]:
```
def test_viterbi(sentences, transition_matrix, emission_matrix, initial_pro
    """
    Returns the accuracy of the Viterbi algorithm on all the sentences.
    """
    accuracies = []
    y_true = []
    y_pred = []
    for sentence in sentences:
        words, true_labels = sentence

        preds_viterbi = viterbi_decoding(words, transition_matrix, emission
        y_true+= true_labels
        y_pred += preds_viterbi
    return get_accuracy(y_true,y_pred)
```

In [34]:
```
def test_viterbi_log(sentences, transition_matrix, emission_matrix, initial
    """
    Returns the average accuracy of the Viterbi algorithm on all the senten
    """
    accuracies = []
    y_true = []
    y_pred = []
    for sentence in sentences:
        words, true_labels = sentence
        preds_viterbi = viterbi_decoding_log(words, transition_matrix, emis
        y_true+= true_labels
        y_pred += preds_viterbi
    return get_accuracy(y_true,y_pred)
```

In [35]:
```
test_viterbi(dev_sentence_set,TM,EM,initial,list(unique_tags))
```

Out[35]: 0.9497753627587882

In [36]:
```
viterbi_accuracy_dev = test_viterbi_log(dev_sentence_set,TM,EM,initial,list
print(f"Accuracy of Viterbi Decoding on dev set: {viterbi_accuracy_dev*100}
```

```
Accuracy of Viterbi Decoding on dev set: 94.97753627587882%
```

Applying **Laplacian Smoothing** to try to increase accuracy:

In [53]:
```
def laplace_smoothing_TM(probability_matrix, k=1e-4):
    return np.array([[(prob + k) / (1 + k * len(probability_matrix[0])) for

def laplace_smoothing_EM(df, k= 1e-10 ):#1e-9
    df_smoothed = df + k
    df_smoothed = df_smoothed.divide(df_smoothed.sum(axis=0) + k * len(df))
    return df_smoothed

EM2 = laplace_smoothing_EM(EM)
TM2 = laplace_smoothing_TM(TM)

viterbi_accuracy_dev2 = test_viterbi_log(make_sentences(dev_data),TM2,EM2,i
print(f"Accuracy of Viterbi Decoding on dev set: {viterbi_accuracy_dev2*100
```

```
Accuracy of Viterbi Decoding on dev set: 94.984366462267%
```

Now, making predictions on the test set:

In [37]:
```
with open('test.json','r') as f:
    test_data = json.load(f)
with open('test.json','r') as f:
    test_data_copy = json.load(f)
```

In [38]:
```
for data_point in test_data:
    for i in range(len(data_point['sentence'])):
        if data_point['sentence'][i] not in vocab:
            data_point['sentence'][i] = pseudo_word(data_point['sentence'][
```

In [39]:
```
# making greedy predictions
greedy_test_preds = [{
    "index":test_data[x]['index'],
    "sentence":test_data_copy[x]['sentence'],
    'labels':greedy_decoding(test_data[x]['sentence'],transition_probabilit
```

In [40]:
```
viterbi_test_preds = [{
    "index":test_data[x]['index'],
    "sentence":test_data_copy[x]['sentence'],
    'labels':viterbi_decoding(test_data[x]['sentence'],TM,EM,initial,list(u
```

Saving the outputs in the desired format

In [102]:
```
with open('greedy.json','w') as f:
    json.dump(greedy_test_preds,f)
```

In [103]:
```
with open('viterbi.json','w') as f:
    json.dump(viterbi_test_preds,f)
```

In [ ]:
```

```

In [ ]: