Task 1: Vocabulary Creation (20 points)

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 >'. Task. Generate a vocabulary from the training data stored in the "train" file and save this vocabulary as "vocab.txt." The format of the "vocab.txt" file should adhere to the following specifications: Each line should consist of a word type, its index within the vocabulary, and its frequency of occurrence, with these elements separated by the tab symbol (\t). The initial line should feature the special token "< unk >," followed by subsequent lines sorted in descending order of occurrence frequency. Please take into account that you are only allowed to use the training data to construct this vocabulary, and you must refrain from using the development and test data.

For instance:

< unk > 0 2000

word1 1 20000

word2 2 10000

Additionally, kindly provide answers to the following questions:

What threshold value did you choose for identifying unknown words for replacement?

What is the overall size of your vocabulary, and how many times does the special token "< unk >" occur following the replacement process?

ANS 1) The threshold value is 2 for replacement. 2) Overall size of vocabulary is 22689 and the token "" appears 19756 times.

```
import json

# Load the training data from the JSON file
with open('data/train.json', 'r') as f:
    train_data = json.load(f)

# Initialize the vocabulary with '<unk>'
vocab = {'<unk>': 0}
threshold = 2

# Iterate through the sentences in the training data
for item in train_data:
    sentence = item['sentence']
    for word in sentence:
        if word.isnumeric():
            word = '<num>'
        if word not in vocab.keys():
```

```
            vocab[word] = 1
        else:
            vocab[word] += 1

# Remove words with frequency less than threshold and make them
unknown
for key in list(vocab.keys())[1:]:
    if vocab[key] < threshold:
        vocab['<unk>'] += vocab[key]
        del vocab[key]

# Sort the vocabulary by frequency
vocab = dict(sorted(vocab.items(), key=lambda x: x[1], reverse=True))

# Ensure that '<unk>' is at the top
if '<unk>' in vocab:
    # Create a new dictionary with '<unk>' as the first item
    new_vocab = {'<unk>': vocab['<unk>']}
    # Add the rest of the vocabulary items
    for word, freq in vocab.items():
        if word != '<unk>':
            new_vocab[word] = freq
    vocab = new_vocab

# Print statistics and save to vocab.txt
print('Threshold:', threshold)
print('Total size of vocabulary:', len(vocab))
print('Count of <unk> tag in vocab:', vocab['<unk>'])

with open('vocab.txt', 'w') as f:
    for index, (key, count) in enumerate(vocab.items()):
        f.write(f'{key}\t{index}\t{count}\n')
```

```
Threshold: 2
Total size of vocabulary: 22689
Count of <unk> tag in vocab: 19756
```

Explanation of the above code:

Loading Data: The code starts by loading training data from a JSON file.

Initializing Vocabulary: It initializes a vocabulary dictionary with a single entry, '', for handling unknown words.

Setting Threshold: A threshold value of 2 is defined to identify rare words.

Processing Sentences: The code processes each sentence in the training data one by one.

Processing Words: For each sentence, it processes individual words. Numeric words are replaced with '', and new words are added to the vocabulary.

Removing Rare Words: Words with counts below the threshold are replaced with '' and removed from the vocabulary.

Sorting Vocabulary: The vocabulary is sorted by word frequency in descending order.

Saving Vocabulary: The final vocabulary is saved to a text file ('vocab.txt') with word, index, and frequency details.

Task 2: Model Learning (20 points)

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:

$t(s'|s) = count(s \to s')/count(s)$

$e(x|s) = count(s \to x)/count(s)$

$\pi(s) = count(null \to s)/count(num\ sentences)$

$t(\cdot|\cdot)$ is the transition parameter.

$e(\cdot|\cdot)$ is the emission parameter.

$\pi(\cdot)$ is the initial state (The sentence begins with this state); also called as prior probabilities.

Task. Learning a model using the training data in the file train and output the learned model into a model file in json format, named hmm.json. The model file should contains two dictionaries for the emission and transition parameters, respectively. The first dictionary, named transition, contains items with pairs of (s,s′) as key and t(s′|s) as value. The second dictionary, named emission, contains items with pairs of (s,x) as key and e(x|s) as value. Additionally, kindly provide answers to the following questions:

How many transition and emission parameters in your HMM?

Ans2) the number of transition parameters is 1392 and the number of emission paramters is 29799.

```
# Load the training data from the JSON file
with open('data/train.json', 'r') as f:
    train_data = json.load(f)
```

```
transition_probabilities = {}
emission_probabilities = {}
state_list = []
label_cnt = {}

# Learn transition and emission parameters from the training data
for item in train_data:
    sentence = item['sentence']
    labels = item['labels']
    prev_state = 'Initial_state'

    for word, label in zip(sentence, labels):
        # Preprocess the word (e.g., handle numbers and unknown words)
        label_cnt[label] = label_cnt.get(label, 0) + 1
        if word.isnumeric():
            word = '<num>'
        if word not in vocab:
            word = '<unk>'
        if label not in state_list:
            state_list.append(label)
        # Update transition parameters
        # if prev_state is not None:
        if str((prev_state, label)) not in
transition_probabilities.keys():
            transition_probabilities[str((prev_state, label))] = 1
        else:
            transition_probabilities[str((prev_state, label))] += 1
        if str((label, word)) not in emission_probabilities.keys():
            emission_probabilities[str((label, word))] = 1
        else:
            emission_probabilities[str((label, word))] += 1
        prev_state = label

total_counts_trans = {}
# Calculate the total counts
for key, value in transition_probabilities.items():
    key_tuple = eval(key)  # Convert the string key back to a tuple
    previous, current = key_tuple  # Unpack the tuple key
    if previous not in total_counts_trans:
        total_counts_trans[previous] = 0
    if previous != 'Initial_state':
        total_counts_trans[previous] = label_cnt[previous]
        transition_probabilities[key] = value /
total_counts_trans[previous]


total_counts_em = {}
for key, value in emission_probabilities.items():
    key_tuple = eval(key)  # Convert the string key back to a tuple
    label, word = key_tuple  # Unpack the tuple key
```

```
    if label not in total_counts_em:
        total_counts_em[label] = 0

    total_counts_em[label] = label_cnt[label]
    emission_probabilities[key] = value / total_counts_em[label]

# Create the HMM model dictionary
hmm_model = {
    'transition': transition_probabilities,
    'emission': emission_probabilities,
}

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

num_transition_params = len(transition_probabilities.keys())
num_emission_params = len(emission_probabilities.keys())

# Print the number of parameters
print(f'Number of transition parameters: {num_transition_params}')
print(f'Number of emission parameters: {num_emission_params}')
```

```
Number of transition parameters: 1392
Number of emission parameters: 29799
```

Explanation of the above code:

Loading Training Data: The code begins by opening and reading a JSON file named 'train.json', which contains training data. This data likely consists of sentences and their corresponding part-of-speech labels.

Initialization: Several variables are initialized:

transition_probabilities and emission_probabilities are dictionaries that will store the learned transition and emission probabilities for the Hidden Markov Model (HMM).

state_list will keep track of all unique part-of-speech labels observed in the training data.

label_cnt is a dictionary that will count the occurrences of each part-of-speech label.

Learning Parameters: The code iterates through each item in the training data. Each item typically represents a sentence and its associated part-of-speech labels.

Preprocessing Words and Labels: For each word and its corresponding label in the sentence, the code preprocesses the word by checking if it's numeric or not in the same way as Task 1. It also tracks the unique labels observed and counts their occurrences.

Learning Transition Probabilities: The code calculates the transition probabilities between part-of-speech labels. It checks if a transition from the previous label (prev_state) to the current label (label) has been observed. If not, it initializes the count to 1; otherwise, it increments the count. The transition probabilities are based on how often transitions occur in the training data.

Learning Emission Probabilities: Similar to transition probabilities, the code calculates emission probabilities between part-of-speech labels and words. It checks if an emission from a label to a word has been observed. If not, it initializes the count to 1; otherwise, it increments the count. Emission probabilities reflect how often a word is emitted from a particular label.

Normalizing Probabilities: After counting the occurrences, the code calculates the normalized probabilities. It divides the counts by the total occurrences of labels to obtain probabilities in the range [0, 1].

Creating the HMM Model: The code organizes the learned transition and emission probabilities into an HMM model dictionary named hmm_model.

Saving to JSON: The HMM model dictionary is saved to a JSON file named 'hmm.json' for later use.

Task 3: Greedy Decoding with HMM (30 points)

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

Task. Implementing the greedy decoding algorithm and evaluate it on the development data. Predict the part-of-speech tags of the sentences in the test data and output the predictions in a file named greedy.json, in the same format of training data. Additionally, kindly provide answers to the following questions:

What is the accuracy on the dev data?

Ans3) The accuracy on the dev data is 93.56%.

```
# Load the HMM model parameters from hmm.json
with open('hmm.json', 'r') as f:
    hmm_model = json.load(f)

# Extract transition and emission probabilities
transition_probabilities = hmm_model['transition']
emission_probabilities = hmm_model['emission']
state_list = state_list
vocab = vocab

# Define a function for greedy decoding
def greedy_decode(sentence, state_list, transition_probabilities,
emission_probabilities, vocab):
    tags = []
    prev_tag = 'Initial_state'

    for token in sentence:
        word = token

        # Preprocess the word (e.g., handle numbers and unknown words)
        if word.isnumeric():
            word = '<num>'
        if word not in vocab:
            word = '<unk>'

        # Initialize the best tag and best score
        best_tag = None
        best_score = 0

        for tag in state_list:
            if tag != 'Initial_state':  # Exclude the 'START' tag
                # Calculate the score for the current tag
                score = transition_probabilities.get(str((prev_tag,
tag)), 1e-6) * emission_probabilities.get(str((tag, word)), 0)

                # Update the best tag if the score is higher
                if score > best_score:
                    best_score = score
                    best_tag = tag

        tags.append(best_tag)
        prev_tag = best_tag

    return tags

# Load the development data from the JSON file
with open('data/dev.json', 'r') as f:
    dev_data = json.load(f)

# Initialize variables for accuracy calculation
total_tokens = 0
```

```
correct_tokens = 0

# Process and decode each sentence in the development data

for item in dev_data:
    sentence = item['sentence']
    gold_labels = item['labels']
    predicted_tags = greedy_decode(sentence, state_list,
transition_probabilities, emission_probabilities, vocab)

    # Calculate accuracy for this sentence
    correct_tokens += sum(1 for gold, pred in zip(gold_labels,
predicted_tags) if gold == pred)
    total_tokens += len(gold_labels)

# Calculate accuracy on the development data
accuracy = correct_tokens / total_tokens
# Print the accuracy
print(f'Accuracy on dev data: {accuracy:.2%}')
```

```
Accuracy on dev data: 93.56%
```

Explanation of the above code:

Loading HMM Model: The code starts by opening and reading a JSON file named 'hmm.json', which contains the previously learned Hidden Markov Model (HMM) parameters, including transition and emission probabilities.

Extracting Model Parameters: It extracts the transition and emission probabilities from the loaded HMM model and assigns them to the variables transition_probabilities and emission_probabilities. The variables state_list and vocab are also passed to the decoding function.

Defining the Greedy Decoding Function: A function named greedy_decode is defined. This function performs the part-of-speech tagging using the learned HMM parameters. It takes the input sentence, the list of possible states (state_list), transition and emission probabilities, and the vocabulary (vocab) as arguments.

Decoding Sentences: The code then loads the development data from the JSON file named 'data/dev.json'. This data likely contains sentences and their corresponding gold labels (correct part-of-speech tags).

Accuracy Calculation Variables: Two variables, total_tokens and correct_tokens, are initialized to calculate the accuracy. total_tokens will keep track of the total number of tokens (words or labels) processed, and correct_tokens will count how many tokens were tagged correctly.

Sentence Decoding and Accuracy Calculation: The code iterates through each item in the development data. For each item, it retrieves the sentence and the gold labels. It then uses the greedy_decode function to predict part-of-speech tags for the sentence based on the learned HMM model.

Calculating Accuracy for Each Sentence: For each sentence, the code calculates how many tokens were correctly tagged by comparing the predicted tags with the gold labels. It increments the correct_tokens variable accordingly.

Final Accuracy Calculation: After processing all sentences, the code calculates the accuracy by dividing the number of correct tokens (correct_tokens) by the total number of tokens in the development data (total_tokens). It prints the accuracy as a percentage.

Task 4: Viterbi Decoding with HMM (30 Points)

The fourth task is to implement the viterbi decoding algorithm with HMM.

Task. Implementing the viterbi decoding algorithm and evaluate it on the development data. Predicting the part-of-speech tags of the sentences in the test data and output the predictions in a file named viterbi.json, in the same format of training data.

Additionally, kindly provide answers to the following questions:

What is the accuracy on the dev data?

Ans4) The accuracy of viterbi decoding on the dev set is 94.83%.

```
import numpy as np
import tqdm

# Load the HMM model parameters from hmm.json
```

```
with open('hmm.json', 'r') as f:
    hmm_model = json.load(f)

# Extract transition and emission probabilities
transition_probabilities = hmm_model['transition']
emission_probabilities = hmm_model['emission']

vocab = vocab
state_list = state_list

def viterbi_decode(sentence, state_list, transition_probabilities,
emission_probabilities, vocab):
    viterbi = np.zeros((len(state_list), len(sentence)),
dtype=np.longdouble)
    tag = np.zeros((len(state_list), len(sentence)), dtype=np.int32)
    predicted_tag = [None] * len(sentence)

    for index, word in enumerate(sentence):
        if word.isnumeric():
            word = '<num>'
        if word not in vocab.keys():
            word = '<unk>'
        if index == 0:
            break

    for i, label in enumerate(state_list):
        viterbi[i, 0] =
transition_probabilities.get(str(('Initial_state', label)), 1e-6) *
emission_probabilities.get(str((label, word)), 0)
        tag[i, 0] = -1

    for index in range(1, len(sentence)):
        token = sentence[index]
        word = token
        if word.isnumeric():
            word = '<num>'
        if word not in vocab.keys():
            word = '<unk>'

        for right, state_right in enumerate(state_list):
            best_score = 0
            max_k = 0
            for left, state_left in enumerate(state_list):
                score = viterbi[left, index-1] *
transition_probabilities.get(str((state_left, state_right)), 1e-6)
                if score > best_score:
                    best_score = score
                    max_k = left
            viterbi[right, index] = best_score *
emission_probabilities.get(str((state_right, word)), 0)
```

```
            tag[right, index] = max_k

    predicted_tag[-1] = state_list[np.argmax(viterbi[:,-1])]

    for i in range(len(sentence) - 2, -1, -1):
        predicted_tag[i] =
state_list[int(tag[state_list.index(predicted_tag[i + 1]), i+1])]

    return predicted_tag

# Load the development data from the JSON file
with open('data/dev.json', 'r') as f:
    dev_data = json.load(f)

# Initialize variables for accuracy calculation
total_tokens = 0
correct_tokens = 0

for item in tqdm.tqdm(dev_data):
    index = item['index']
    sentence = item['sentence']
    gold_labels = item['labels']
    predicted_tags = viterbi_decode(sentence, state_list,
transition_probabilities, emission_probabilities, vocab)

    # Calculate accuracy for this sentence
    correct_tokens += sum(1 for gold, pred in zip(gold_labels,
predicted_tags) if gold == pred)
    total_tokens += len(gold_labels)

# Calculate accuracy on the development data
accuracy = correct_tokens / total_tokens
# Print the accuracy
print(f'Accuracy on dev data: {accuracy:.2%}')
```

```
100%|██████████| 5527/5527 [03:47<00:00, 24.31it/s]
```

```
Accuracy on dev data: 94.83%
```

Explanation of the above code:

Loading HMM Model: The code starts by opening and reading a JSON file named 'hmm.json', which contains the previously learned Hidden Markov Model (HMM) parameters, including transition and emission probabilities.

Extracting Model Parameters: It extracts the transition and emission probabilities from the loaded HMM model and assigns them to the variables transition_probabilities and emission_probabilities. The variables vocab and state_list are also passed to the decoding function.

Defining the Viterbi Decoding Function: A function named viterbi_decode is defined. This function performs part-of-speech tagging using the Viterbi algorithm based on the learned HMM parameters. It takes the input sentence, the list of possible states (state_list), transition and emission probabilities, and the vocabulary (vocab) as arguments.

Viterbi Decoding: Inside the viterbi_decode function, a Viterbi matrix (viterbi) and a tag matrix (tag) are initialized to store intermediate results. A list named predicted_tag is initialized to store the predicted part-of-speech tags for each token in the sentence.

Preprocessing First Token: The code preprocesses the first token in the sentence by checking if it's numeric or not in the same way as earlier code. This is done only once for efficiency.

Initialization Step: The Viterbi matrix's first column is initialized based on the transition probabilities from the 'Initial_state' to each state and the emission probability of the first word.

Dynamic Programming Step: The code iterates through the remaining columns of the Viterbi matrix, filling in each cell with the highest probability score. It uses dynamic programming to compute the scores efficiently.

Backtracking for Predicted Tags: After computing the Viterbi matrix, the code performs backtracking to determine the most likely sequence of part-of-speech tags (the predicted_tags). Starting from the last token, it traces back to find the most likely tag for each token in the sentence.

Accuracy Calculation: The code loads the development data and, for each sentence, predicts part-of-speech tags using Viterbi decoding. It then calculates how many of these predictions match the gold labels (correct_tokens) and the total number of tokens processed (total_tokens).

Final Accuracy: After processing all sentences, the code computes the accuracy by dividing the number of correct tokens by the total number of tokens and expresses it as a percentage. This accuracy score indicates the model's performance on the development data.

In the below code cells we just iterate through the test json and create the greedy and viterbi json files using the above functions.

```
with open('data/test.json', 'r') as f:
    test_data = json.load(f)

decoded_data_greedy = []

for item in test_data:
    index = item['index']
    sentence = item['sentence']

    predicted_tags = greedy_decode(sentence, state_list,
```

```
transition_probabilities, emission_probabilities, vocab)
    decoded_data_greedy.append({'index': item['index'], 'sentence':
sentence, 'labels': predicted_tags})

# Output the predictions to greedy.json
with open('greedy.json', 'w') as f:
    json.dump(decoded_data_greedy, f, indent=4)

with open('data/test.json', 'r') as f:
    test_data = json.load(f)

decoded_data_viterbi = []

for item in tqdm.tqdm(test_data):
    index = item['index']
    sentence = item['sentence']

    predicted_tags = viterbi_decode(sentence, state_list,
transition_probabilities, emission_probabilities, vocab)
    decoded_data_viterbi.append({'index': item['index'], 'sentence':
sentence, 'labels': predicted_tags})

# Output the predictions to greedy.json
with open('viterbi.json', 'w') as f:
    json.dump(decoded_data_viterbi, f, indent=4)
```

```
100%|██████████| 5462/5462 [03:40<00:00, 24.75it/s]
```