In [1]:
```
import json
from collections import Counter, defaultdict
import numpy as np
```

In [2]:
```
# Utility function
def replace_infrequent_words(data):
    for entry in data:
        entry['sentence'] = [word if word in vocab_list else '<unk>' for word in entry['sentence']]
    return data

def load_model(file_name):
    with open(file_name, 'r') as f:
        json_model = json.load(f)
    return json_model
```

In [3]:
```
train_data = load_model('data/train.json')
label_list = list(set(label for record in train_data for label in record['labels']))
```

Task 1: Vocabulary Creation

In [4]:
```
def generate_vocabulary():
    word_list = []
    threshold = 2

    for data_dict in train_data:
        word_list.extend(data_dict.get("sentence"))

    word_dict = Counter(word_list)


    # Replacing rare words whose occurences are less than the threshold value 2
    for word in list(word_dict.keys()):
        if word_dict[word] <= threshold:
            word_dict["<unk>"] += word_dict[word]
            del word_dict[word]


    # Sort by frequency
    vocabs = sorted(word_dict.items(), key = lambda x: x[1], reverse = True)

    with open("verification/out/vocab.txt", "w") as file:
        count_unk = word_dict["<unk>"]
        file.write(f"<unk>\t{0}\t{count_unk}\n")
        for index, (word, frequency) in enumerate(vocabs, start=1):
            if word != '<unk>':
                file.write(f"{word}\t{index}\t{int(frequency)}\n")

    vocab_list = {k: v for i, (k,v) in enumerate(vocabs)}
    return vocab_list
```

In the above code I initialize the word list and threshold. I collect words from training data, count the frquency of each word. After this step I replace the rare words with the tag . Finally I sort the the values based on frequency and store in in a text file.

In [5]:
```
vocab_list = generate_vocabulary()
train_data = replace_infrequent_words(train_data)
```

In [11]:
```
print(len(vocab_list))
print(vocab_list.get('<unk>'))
```

```
16920
32537
```

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?

- The threshold value used is 2
- The overall vocabolary consists of 16920 values
- The special token occurs 32537 times

Task 2: Model Learning

In [6]:
```
transition = defaultdict(int)
emission = defaultdict(int)
initial_state = defaultdict(int)
state_freq = defaultdict(int)
sentences = 0
for data in train_data:
    word_list = data["sentence"]
    labels = data["labels"]
    sentences += 1
    prev_state = None
    for word, label in zip(word_list, labels):
        if prev_state is not None:
            transition[(prev_state, label)] += 1
        else:
            initial_state[label] += 1
        emission[(label, word)] += 1
        state_freq[label] += 1
        prev_state = label
transition_probab = {str(k): v/state_freq[k[0]] for k, v in transition.items()}
emission_probab = {str(k): v/state_freq[k[0]] for k, v in emission.items()}
initial_start_probab = {str(k): v/sentences for k, v in initial_state.items()}
model = {'transition': transition_probab, 'emission': emission_probab, 'initial_state': initial_start_probab}
with open('verification/out/hmm.json', 'w') as f:
        json.dump(model, f)
```

The above code is used to calculate the transition, emission and initial state probabilities. For the dataset provided I go through each word in each sentence. If the word has not occured previously, I increment the counter for the initial state, else I increment the counter for the transition list. Finally using these values I calculate the transition and emission probabilities by dividing it with the total frequencies. To summarize:

- The transition probability is the probability of transitioning from one label to another
- The emisiion probablity refers to the probability of an observed state given a hidden state
- The initial_state keeps track of the number of times each label appears as the first label in a sentence in your training data

In [12]:
```
print(len(transition_probab))
print(len(emission_probab))
```

```
1351
23373
```

Additionally, kindly provide answers to the following questions: How many transition and emission parameters in your HMM?

- There are 1351 transition probabilities
- There are 23373 emission probabilities

Greedy Algorithm

In [7]:

```
def evaluateGreedy(test_data):
    correct = total = 0
    for entry in test_data:
        sentence = entry['sentence']
        originalSentence = sentence
        sentence = [word if word in vocab_list else '<unk>' for word in sentence]
        labels = entry['labels']
        predictions = greedy(sentence)
        correct += sum(p == l for p, l in zip(predictions, labels))
        total += len(labels)
    return correct / total

def get_greedy_prediction_labels(test_data):
    json_result = []
    for entry in test_data:
        sentence = entry['sentence']
        originalSentence = sentence
        sentence = [word if word in vocab_list else '<unk>' for word in sentence]
        predictions = greedy(sentence)
        json_result.append({
            "index": entry['index'],
            "sentence": originalSentence,
            "labels": predictions
        })
    with open('verification/out/greedy.json', 'w') as f:
        json.dump(json_result, f)

def greedy(sentence):
    hmm_model = load_model('verification/out/hmm.json')
    initial_state = hmm_model['initial_state']
    transition = hmm_model['transition']
    emission = hmm_model['emission']
    y = [None] * len(sentence)
    y[0] = max(label_list, key=lambda s: initial_state.get(s,0) * emission.get(str((s, sentence[0])),0))
    for i in range(1, len(sentence)):
        y[i] = max(label_list, key =lambda s: transition.get(str((y[i-1], s)), 0) * emission.get(str((s,
sentence[i])),0))
    return y
```

1. In the above code the **greedy** function performs the major task. This function implements the greedy decoding algorithm for a given sentence. It first loads the HMM model from a JSON file, which includes initial state probabilities, transition probabilities, and emission probabilities that was calculated in step 2. It then initializes the first label in the sequence as the label that maximizes the product of the initial state probability and emission probability for the first word. For each subsequent word, it selects the label that maximizes the product of the transition probability from the previous label and emission probability for that word.
2. The evaluateGreedy is used to evaluate the accuracy of the greedy algorithm given that the input test data also contains the labels
3. The get_greedy_prediction_labels just returns the predicted labels for the sentences in the given dataset

In [15]:
```
# #  Loading Dev & test data
test_data = load_model('data/test.json')
dev_data = load_model('data/dev.json')


get_greedy_prediction_labels(test_data)
```

What is the accuracy on the dev data?

In [16]:
```
accuracy = evaluateGreedy(dev_data)
print(f'Accuracy: {accuracy}')
```

```
Accuracy: 0.9298084512172909
```

The accuracy on the Dev data is 92.98%

Viterbi algorithm

In [23]:
```
labels = list(set(label for record in train_data for label in record['labels']))

# Evaluates the accuracy score of the algorithm
def evaluate_accuracy(data, all_paths):
    correct = 0
    total = 0
    for i in range(len(data)):
        labels = data[i]['labels']
        path = all_paths[i]
        for j in range(len(labels)):
            if labels[j] == path[j]:
                correct += 1
            total += 1
    return correct / total


def viterbi_decode(data, model):
    initial = np.array([model['initial_state'].get(state, 0) for state in labels])
    transition = np.array([[model['transition'].get(str((state1, state2)), 0) for state2 in labels] for state1
in labels])

    viterbi_paths = []
    viterbi_json = []

    for sentence_data in data:

        sentence = sentence_data['sentence']
        original_sentence = sentence

        sentence = [word if word in vocab_list else '<unk>' for word in sentence]
        emission = np.array([[model['emission'].get(str((state, word)),0) for word in sentence] for state in
labels])

        # initializing and filling the first columns of viterbi
        viterbi = np.zeros((len(transition), len(sentence)))
        viterbi[:, 0] = initial * emission[:, 0]

        #Initializing the backpointer
        backpointers = np.zeros((len(initial), len(sentence)), 'int')

        for t in range(1, len(sentence)):
            for s in range(len(initial)):
                trans_probs = viterbi[:, t-1] * transition[:, s]
```

```
                max_prob = np.max(trans_probs)
                viterbi[s, t] = max_prob * emission[s, t]
                backpointers[s, t] = np.argmax(trans_probs)


        path = []
        current_state = np.argmax(viterbi[:, -1])
        for t in range(len(sentence)-1, -1, -1):
            path.append(labels[current_state])
            current_state = backpointers[current_state, t]
        path = path[::-1]

        viterbi_json.append({
            "index": sentence_data['index'],
            "sentence": original_sentence,
            "labels": path
        })

        viterbi_paths.append(path)

    with open('verification/out/viterbi.json', 'w') as f:
        json.dump(viterbi_json, f)

    return viterbi_paths

def load_model(file_path):
    with open(file_path, 'r') as f:
        model = json.load(f)
    return model

model = load_model('verification/out/hmm.json')
```

The above code contains majorly 2 fuctions:
- evaluate_accuracy: It evaluates the accuracy of the predictions - viterbi_decode: This function implements the Viterbi algorithm, which is used to find the most likely sequence of hidden states given a sequence of observations

For a sentence the viterbi_decode does as follows:<\b>

- It replaces any unknown word not in the vocabulary list with
- It creates an emission matrix, which represents the probability of observing each word given each state. It also initializes Viterbi matrix and backpointer.

- The Viterbi matrix stores the maximum probability of reaching each state at each time step, while the backpointer matrix stores the state that maximized this probability. For each time step and each state, it calculates the maximum probability of reaching that state at that time step.
- After filling the matrices, the backpointer is used to find the most likely sequence of state
- The output is finally dumped into viterbi.json

What is the accuracy on the dev data?

In [20]:
```
all_paths_dev = viterbi_decode(dev_data, model)
accuracy = evaluate_accuracy(dev_data, all_paths_dev)
print(f'Accuracy: {accuracy}')
```

```
Accuracy: 0.9437875660251351
```

```
The accuracy on test data is 94.3%
```

In [22]:
```
test_data = load_model('data/test.json')
all_paths = viterbi_decode(test_data, model)
```

In [ ]:
```

```