```
import pandas as pd
import numpy as np
import json
```

Task 1: Vocabulary Creation

In task 1, we first design a function to replace rare words whose occurence is less than the set threshold value with a special token . We use this function on the training data and store all unique words in a dataframe called vocab and arrange them in descending order of their occurence. Next, we find the rows which have the special token and move it to the top, while storing the index of each word in the vocab dataframe. Finally, we create the vocab.txt file which contains this vocabulary from training data.

```
df = pd.read_csv("data/train", sep = "\t", names = ['id', 'words', 'pos'])
df['occ'] = df.groupby('words')["words"].transform('size')

threshold = 2
def word_replace(row):
    if row.occ < threshold:
        return "<unk>"
    else:
        return row.words

df['words'] = df.apply(lambda row : word_replace(row), axis = 1)
vocab = df.words.value_counts().rename_axis('words').reset_index(name = 'occ')

unk = vocab[vocab['words'] == "<unk>"]

index = vocab[vocab.words == "<unk>"].index
vocab = vocab.drop(index)
vocab = pd.concat([unk, vocab]).reset_index(drop = True)
vocab['id'] = vocab.index + 1
cols = vocab.columns.tolist()
cols = [cols[0], cols[-1], cols[1]]
vocab = vocab[cols]

unk_count = int(vocab[vocab["words"] == "<unk>"].occ)

vocab.to_csv("vocab.txt", sep="\t", header=None)

print("Threshold for replacing rare words:", threshold)
print("Size of vocabulary:",vocab.shape[0])
print("The total occurences of the special token <unk>:", unk_count)
```

```
Threshold for replacing rare words: 2
Size of vocabulary: 23183
The total occurences of the special token <unk>: 20011
```

Preprocessing the train data into required format for the upcoming tasks. First, we collect all the unique tags from the train data. Then, we create a nested list for storing the training data where each list has a tuple corresponding to a row in train data

```
pos = df.pos.value_counts().rename_axis('pos').reset_index(name = 'count')
tags = pos.pos.tolist()

sentences = []
sentence = []
first = 1
for line in df.itertuples():
    if(line.id == 1 and first == 0):
        sentences.append(sentence)
        sentence = []
    first = 0
    sentence.append((line.words, line.pos))
sentences.append(sentence)
```

Task 2: Model Learning

In task 2, trans_matrix function computes the transition matrix for the training data using the given formula, and emission_matrix function computes the emission matrix for the given sentences in traing data. trans_prob function converts the transition matrix into a transition dictionary, and emission_prob function converts the emission matrix into an emission dictionary.

```
def trans_matrix(sentences, tags):
    tag_cnt = {}
    t_matrix = np.zeros((len(tags),len(tags)))
    for tag in range(len(tags)):
        tag_cnt[tag] = 0

    for sentence in sentences:
        for i in range(len(sentence)):
            tag_cnt[tags.index(sentence[i][1])] += 1
            if i == 0:
                continue
            t_matrix[tags.index(sentence[i - 1][1])][tags.index(sentence[i][1])] += 1

    for i in range(t_matrix.shape[0]):
        for j in range(t_matrix.shape[1]):
            if(t_matrix[i][j] == 0) : t_matrix[i][j] = 1e-10
            else: t_matrix[i][j] /= tag_cnt[i]

    return t_matrix


def emission_matrix(tags, vocab, sentences):
    tag_cnt = {}
    e_matrix = np.zeros((len(tags), len(vocab)))
    for tag in range(len(tags)):
        tag_cnt[tag] = 0

    for sentence in sentences:
        for word, pos in sentence:
            tag_cnt[tags.index(pos)] +=1
            e_matrix[tags.index(pos)][vocab.index(word)] += 1

    for i in range(e_matrix.shape[0]):
        for j in range(e_matrix.shape[1]):
            if(e_matrix[i][j] == 0) : e_matrix[i][j] = 1e-10
            else: e_matrix[i][j] /= tag_cnt[i]

    return e_matrix

vocab = vocab.words.tolist()


def trans_prob(tags, t_matrix, prior_prob):
    tags_dict = {}

    for i, tags in enumerate(tags):
        tags_dict[i] = tags

    trans_prob = {}
    for i in range(t_matrix.shape[0]):
        trans_prob['(' + '<\S>' + ',' + tags_dict[i] + ')'] = prior_prob[tags_dict[i]]
    for i in range(t_matrix.shape[0]):
        for j in range(t_matrix.shape[1]):
            trans_prob['(' + tags_dict[i] + ',' + tags_dict[j] + ')'] = t_matrix[i][j]


    return trans_prob

def emission_prob(tags, vocab, e_matrix):
    tags_dict = {}

    for i, tags in enumerate(tags):
        tags_dict[i] = tags

    emission_prob = {}

    for i in range(e_matrix.shape[0]):
        for j in range(e_matrix.shape[1]):
            emission_prob['(' + tags_dict[i] + ', ' + vocab[j] + ')'] = e_matrix[i][j]

    return emission_prob
```

Here, get_all_prob function generates the transition matrix, emission matrix, transition dictionary and the emission dictionary, inital_prob function calculates the initial transition probability for each tag and we store the transition and emission dictionaries in a json file named 'hmm.json'

```
def get_all_prob(tags, vocab, sentences, prior_prob):
    t_matrix = trans_matrix(sentences, tags)
    e_matrix = emission_matrix(tags, vocab, sentences)

    transition_probability = trans_prob(tags, t_matrix, prior_prob)
    emission_probability = emission_prob(tags, vocab, e_matrix)

    return transition_probability, emission_probability

def inital_prob(df, tags):
    tags_start_cnt = {}
    total_start_sum = 0
    for tag in tags:
        tags_start_cnt[tag] = 0

    for line in df.itertuples():
        if(line[1] == 1):
            tags_start_cnt[line[3]]+=1
            total_start_sum += 1

    prior_prob = {}
    for key in tags_start_cnt:
        prior_prob[key] = tags_start_cnt[key] / total_start_sum

    return prior_prob

prior_prob = inital_prob(df, tags)
trans_prob, emission_prob = get_all_prob(tags, vocab, sentences, prior_prob)


print("Number of Transition Parameters:",len(trans_prob))
print("Number of Emission Parameters:",len(emission_prob))

with open('hmm.json', 'w') as f:
    json.dump({"transition": trans_prob, "emission": emission_prob}, f, ensure_ascii=False, indent = 4)
```

```
Number of Transition Parameters: 2070
Number of Emission Parameters: 1043235
```

Task 3: Greedy Decoding with HMM

In task 3, we read the dev file and create a nested list for storing the development data where each list has a tuple corresponding to a row in dev data
Implement greedy_decoding function which computes the state sequence for HMM Model using the greedy decoding technique
evaluate function computes the accuracy of the model model by comparing the the predicted tag sequence with groundtruth

```
dev_data = pd.read_csv("data/dev", sep = '\t', names = ['id', 'words', 'pos'])
dev_data['occ'] = dev_data.groupby('words')['words'].transform('size')

valid_sentences = []
sentence = []
first = 1
for line in dev_data.itertuples():
    if(line.id == 1 and first == 0):
        valid_sentences.append(sentence)
        sentence = []
    first = 0
    sentence.append((line.words, line.pos))
valid_sentences.append(sentence)

def greedy_decoding(trans_prob, emission_prob, prior_prob, valid_sentences, tags):
    sequences = []
    total_score = []
    for sentence in valid_sentences:
        prev_tag = None
        sequence = []
        score = []
```

```
        for i in range(len(sentence)):
            best_score = -1
            for j in range(len(tags)):
                state_score = 1
                if i == 0:
                    state_score *= prior_prob[tags[j]]
                else:
                    if str("(" + prev_tag  + "," + tags[j] + ")") in trans_prob:
                        state_score *= trans_prob["(" + prev_tag  + "," + tags[j] + ")"]

                if str("(" + tags[j] + ", " + sentence[i][0] + ")") in emission_prob:
                    state_score *= emission_prob["(" + tags[j] + ", " + sentence[i][0] + ")"]
                else:
                    state_score *= emission_prob["(" + tags[j] + ", " + "<unk>" + ")"]

                if(state_score > best_score):
                    best_score = state_score
                    highest_prob_tag = tags[j]

            prev_tag = highest_prob_tag
            sequence.append(prev_tag)
            score.append(best_score)
        sequences.append(sequence)
        total_score.append(score)

    return sequences, total_score

sequences, total_score = greedy_decoding(trans_prob, emission_prob, prior_prob, valid_sentences, tags)


def evaluate(sequences, valid_sentences):
    total = 0
    correct = 0
    for i in range(len(valid_sentences)):
        for j in range(len(valid_sentences[i])):

            if(sequences[i][j] == valid_sentences[i][j][1]):
                correct += 1
            total +=1

    accuracy = correct / total
    return accuracy

print('Accuracy of Greedy Decoding HMM model on dev data: {}'.format(evaluate(sequences, valid_sentences)*100))
```

```
Accuracy of Greedy Decoding HMM model on dev data: 93.51132293121243
```

Here, we read the test data and create a nested list for storing the testing data where each list has a tuple corresponding to a row in test data
The output_file function stores the predictions of pos tags using greedy decoding by the HMM model on the test data in 'greedy.out'

```
test_data = pd.read_csv("data/test", sep = '\t', names = ['id', 'words'])
test_data['occ'] = test_data.groupby('words')['words'].transform('size')
test_data['words'] = test_data.apply(lambda row : word_replace(row), axis = 1)

test_sentences = []
sentence = []
first = 1
for line in test_data.itertuples():
    if(line.id == 1 and first == 0):
        test_sentences.append(sentence)
        sentence = []
    first = 0
    sentence.append(line.words)
test_sentences.append(sentence)

test_sequences, test_score = greedy_decoding(trans_prob, emission_prob, prior_prob, test_sentences, tags)


def output_file(test_inputs, test_outputs, filename):
    result = []
    for i in range(len(test_inputs)):
        s = []
        for j in range(len(test_inputs[i])):
            s.append((str(j+1), test_inputs[i][j], test_outputs[i][j]))
        result.append(s)
```

```
    with open(filename + ".out", 'w') as f:
        for element in result:
            f.write("\n".join([str(item[0]) + "\t" + item[1] + "\t" + item[2] for item in element]))
            f.write("\n\n")

output_file(test_sentences, test_sequences, "greedy")
```

> **Task 4: Viterbi Decoding with HMM**

In task 4, we implement the viterbi decoding algorithm using viterbi_decoding function on dev data which computes the probabilty for each word in a sentence having a tag from the group of all tags
viterbi_backward function finds the best possible tag sequence for each sentence based on the probabilites calculated by the viterbi_decoding function

```
def viterbi_decoding(trans_prob, emission_prob, prior_prob, sentence, tags):

    n = len(tags)
    viterbi_list = []
    data = {}
    for t in tags:
        if str("(" + t + ", " + sentence[0][0] + ")") in emission_prob:
            viterbi_list.append(prior_prob[t] * emission_prob["(" + t + ", " + sentence[0][0] + ")"])
        else:
            viterbi_list.append(prior_prob[t] * emission_prob["(" + t + ", " + "<unk>" + ")"])

    for i, l in enumerate(sentence):
        word = l[0]
        if i == 0: continue
        temp_list = [None] * n
        for j,tag in enumerate(tags):
            score = -1
            val = 1
            for k, prob in enumerate(viterbi_list):
                if str("(" + tags[k] + "," + tag + ")") in trans_prob and str("(" + tag + ", " + word + ")") in emission_prob:
                    val = prob * trans_prob["(" + tags[k] + "," + tag + ")"] * emission_prob["(" + tag + ", " + word + ")"]
                else:
                    val = prob * trans_prob["(" + tags[k] + "," + tag + ")"] * emission_prob["(" + tag + ", " + "<unk>" + ")"]
                if(score < val):
                    score = val
                    data[str(i) + ", " + tag] = [tags[k], val]
            temp_list[j] = score
        viterbi_list = [x for x in temp_list]

    return data, viterbi_list


c = []
v = []
for sentence in valid_sentences:
    a, b = viterbi_decoding(trans_prob, emission_prob, prior_prob, sentence, tags)
    c.append(a)
    v.append(b)

def viterbi_backward(tags, data, viterbi_list):

    num_states = len(tags)
    n = len(data) // num_states
    best_sequence = []
    best_sequence_breakdown = []
    x = tags[np.argmax(np.asarray(viterbi_list))]
    best_sequence.append(x)

    for i in range(n, 0, -1):
        val = data[str(i) + ', ' + x][1]
        x = data[str(i) + ', ' + x][0]
        best_sequence = [x] + best_sequence
        best_sequence_breakdown =  [val] + best_sequence_breakdown

    return best_sequence, best_sequence_breakdown

best_seq = []
best_seq_score = []
for data, viterbi_list in zip(c, v):
```

```
    a, b = viterbi_backward(tags, data, viterbi_list)
    best_seq.append(a)
    best_seq_score.append(b)

print('Accuracy of Viterbi Decoding HMM model on dev data: {}'.format(evaluate(best_seq, valid_sentences)*100))
```

```
Accuracy of Viterbi Decoding HMM model on dev data: 94.80905834496994
```

Here, we use the viterbi decoding algorithm for predicting the pos tags of all sentences in the test data
We use the output_file function to store the predictions of Viterbi decoding by the HMM model on the test data in 'viterbi.out'

```
c = []
v = []
for sentence in test_sentences:
    a, b = viterbi_decoding(trans_prob, emission_prob, prior_prob, sentence, tags)
    c.append(a)
    v.append(b)

best_seq = []
best_seq_score = []
for data, viterbi_list in zip(c, v):
    a, b = viterbi_backward(tags, data, viterbi_list)
    best_seq.append(a)
    best_seq_score.append(b)
output_file(test_sentences, best_seq, 'viterbi')
```

```
#References:
#https://stackoverflow.com/questions/36656870/replace-rare-word-tokens-python
#https://github.com/ananthpn/pyhmm
#https://www.katrinerk.com/courses/python-worksheets/hidden-markov-models-for-pos-tagging-in-python
#https://github.com/ngoquanghuy99/Hidden-Markov-Models-for-POS-Tagging
#https://en.wikipedia.org/wiki/Viterbi_algorithm
```