Import libraries

In [164]:
```
import csv
import time
import json
import os
```

Task 1 - Vocabulary Creation

The data folder containing train, dev and text files should be present in same location as this ipynb to run it. The below cell reads train data to prepare vocab.txt

Also prepared dev and test data set for evaluation.

vocabulary is created only using training data

In [165]:
```
vocab = dict()

directory = os.getcwd()
url_train = os.path.join(directory, "data/train")
url_dev = os.path.join(directory, "data/dev")
url_test = os.path.join(directory, "data/test")

with open(url_train, "r", encoding="utf8") as train_file:
    train_reader = csv.reader(train_file, delimiter="\t")

    sent = []
    train_set = []

    for row in train_reader:
        if row == []:
            train_set.append(sent)
            sent = []
        else:
            (index, word_type, pos_tag) = row

            sent.append((word_type, pos_tag))

            if word_type not in vocab:
                vocab[word_type] = 1
            else:
                vocab[word_type] += 1


with open(url_dev, "r", encoding="utf8") as file:
    dev_reader = csv.reader(file, delimiter="\t")

    sent_dev = []
    dev_set = []

    for row in dev_reader:
        if row == []:
            dev_set.append(sent_dev)
            sent_dev = []
        else:
            (index, word, pos_tag) = row

            sent_dev.append((word, pos_tag))


with open(url_test, "r", encoding="utf8") as file:
    test_reader = csv.reader(file, delimiter="\t")

    sent_test = []
    test_set = []

    for row in test_reader:
        if row == []:
            test_set.append(sent_test)
            sent_test = []
        else:
            (index, word) = row

            sent_test.append(word)
```

we count the occurence of unique words in train and also replace all words with frequency less than or equal to 3 with "< unk >" tag.

In [166]:
```
threshold = 3
sum = 0
toremove = []
for key in vocab.keys():
    if vocab[key] < threshold:
        sum += vocab[key]
        toremove.append(key)

vocab['< unk >'] = sum
```

Order the vocabulary in decresing order

In [167]:
```
converted_vocab = dict(sorted(vocab.items(), key=lambda x:x[1], reverse=True))
```

Writing the output of vocabulary into a txt file named vocab.txt. The format of the vocabulary file is that each line contains a (word type, index, occurrences) separated by the tab '\t'. The first line is the special token "< unk >" and the following lines are sorted by its occurrences in descending.

In [168]:
```
j = 1
url_vocab = os.path.join(directory, "vocab.txt")
with open(url_vocab, 'a') as f:
    f.write('< unk >' + '\t' + str(0) + '\t' + str(converted_vocab['< unk >']) + '\n')
    converted_vocab.pop('< unk >')
    for i in converted_vocab:
        f.write(i + "\t" + str(j) + "\t" + str(converted_vocab[i]) + "\n")
        j += 1
```

In [169]:
```
vocab['< unk >']
```

Out[169]:
```
32537
```

In [178]:
```
len(vocab)
```

Out[178]:
```
43194
```

Selected threshold for unknown words replacement : 3

Total size of vocabulary : 43194 (including "< unk >" tag)

Total occurences of special token '< unk >' after replacement : 32537

Task 2 - Model Learning

In [170]:
```
# list of tagged words
dev_set_combined = [tup for sent in dev_set for tup in sent]

# list of untagged words
dev_set_words = [tup[0] for sent in dev_set for tup in sent]
```

To calculate transition and emission probabilities I am initially finding the count of unique (tag1, tag2) pairs and (word1, tag1) pairs for transition and emission respectively.

For transition "tag2" is followed by "tag1".

For emission "word1" is assigned "tag1"

Also counting unique "tag" frequencies

In [171]:
```
train_set_combined = [tup for sent in train_set for tup in sent]
tag_counts = dict()
emission_counts = dict()
transition_counts = dict()

prev_tag = '.'

for tup in train_set_combined:

    if tup not in emission_counts:
        emission_counts[tup] = 1
    else:
        emission_counts[tup] += 1

    if (prev_tag, tup[1]) not in transition_counts:
        transition_counts[(prev_tag, tup[1])] = 1
    else:
        transition_counts[(prev_tag, tup[1])] += 1

    if tup[1] not in tag_counts:
        tag_counts[tup[1]] = 1
    else:
        tag_counts[tup[1]] += 1

    prev_tag = tup[1]
```

Calculating emission probabilities

Emission probability = (frequency of unique (word1, tag1)) / (frequency of tag1)

In [172]:
```
emission_prob = dict()
for pair in emission_counts:
    emission_prob[pair] = emission_counts[pair]/tag_counts[pair[1]]
```

Calculating transition probabilities

Transition probability = (frequency of unique (tag1, tag2)) / (frequency of tag1)

In [173]:
```
transition_prob = dict()
for pair in transition_counts:
    transition_prob[pair] = transition_counts[pair]/tag_counts[pair[0]]
```

In [174]:
```
t_dict = dict()
for key in transition_prob:
    t_dict[str(key)] = transition_prob[key]

e_dict = dict()
for key in emission_prob:
    key_r = (key[1], key[0])
    e_dict[str(key_r)] = emission_prob[key]
```

Creating hmm.json with transition and emission probabilities dictionary. Probabilites are generated only over training data. The hmm.json contains dictionary, with key named transition, whose value contains items with pairs of (s,s′) as key and t(s′|s) as value. The second key, named emission, contains items with pairs of (s, x) as key and e(x|s) as value.

In [175]:
```
hmm = {}
hmm['transition_probabilities'] = t_dict
hmm['emission_probabilities'] = e_dict

url_hmm = os.path.join(directory, "hmm.json")

with open(url_hmm, "a") as outfile:
    json.dump(hmm, outfile)
```

In [176]:
```
len(t_dict)
```

Out[176]: 1378

In [177]:
```
len(e_dict)
```

Out[177]: 50285

Number of transition parameters : 1378

Number of emission parameters : 50285

Task 3 - Greedy Decoding with HMM

In [179]:
```
T = []
for i in tag_counts:
    if i not in T:
        T.append(i)
```

I am considering '.' tag as the start tag of every sentence. Also calculated emission and transition probabilities with '.'

Greedy algorithm takes list of words separated with '.' and finds respective tags. It returns list of (word, tag) pairs.

Given the list of words - for every iteration greedy algorithm finds trasition probability for (tag1, tag2) and emission for (word1, tag1). Calculates state probability which is transition * emission. Does this for all 45 tags for every word. For each word we find the tag with maximum state probability and assign that tag to that word.

In [180]:
```
# words - list of words
def Greedy(words):
    state = [] # to store list of tag assigned to each word

    for key, word in enumerate(words): # iterate over each word to find tag
        p = []
        for tag in T: # for each word find state probability of every tag and pick the maximum
            if key == 0:
                if ('.', tag) not in transition_prob: # give trasition probability = 0 if pair not found. Initially we
                    transition_p = 0
                else:
                    transition_p = transition_prob[('.', tag)]
            else:
                if (state[-1], tag) not in transition_prob:
                    transition_p = 0
                else:
                    transition_p = transition_prob[(state[-1], tag)]

            if (word, tag) not in emission_prob: # give emission probability = 0 if pair not found
                emission_p = 0
            else:
                emission_p = emission_prob[(word, tag)]

            state_probability = emission_p * transition_p
            p.append(state_probability) # list of state probabilities for a word-tag combination

        pmax = max(p) # find maximum probability

        state_max = T[p.index(pmax)] # find tag with that maximum state probability
        state.append(state_max)
    return list(zip(words, state))
```

Calculating accuracy on dev data set using greedy algorithm.

In [181]:
```
tagged_seq = Greedy(dev_set_words)

# accuracy
check = [i for i, j in zip(tagged_seq, dev_set_combined) if i == j]

accuracy = len(check)/len(tagged_seq)
print('Greedy Algorithm Accuracy: ',accuracy*100)
```

```
Greedy Algorithm Accuracy:  93.78296938922665
```

In [182]:
```
# list of tagged words
test_set_combined = [tup for sent in test_set for tup in sent]
```

Finding tags for test dataset

In [183]:
```
test_tagged_seq = Greedy(test_set_combined)
```

Writing test dataset output to greedy.out

In [184]:
```
j = 1

url_greedy = os.path.join(directory, "greedy.out")

with open(url_greedy, 'a') as f:
    for tup in test_tagged_seq:
        if tup[0] == '.':
            f.write(str(j) + "\t" + tup[0] + "\t" + tup[1] + "\n")
            f.write("\n")
            j = 0
        else:
            f.write(str(j) + "\t" + tup[0] + "\t" + tup[1] + "\n")
        j += 1
```

Accuracy of Greedy algorithm on Dev set : 93.783 %

Task 4 - Viterbi Decoding with HMM

considering '.' as start tag. Give list of words separated by '.' to viterbi algorithm. We create Viterbi (to calculate state probabilities for each word-tag combination and previous tag probability) and backpointer (to find the tag which gave the maximum probability to present tag) matrices.

in viterbi the state probability = (previous tag state probability) * (trasition probability) * (emission probability)

In viterbi we dont compute the probability for every combination of tags possible for a sentence, instead we store previous sequence of tag probablities.

In [185]:
```python
def viterbi_algorithm(observation):
    n = len(T)
    t = len(observation)
    viterbi = [[0] * t for _ in range(n)]
    backpointer = [[0] * t for _ in range(n)]
    all_tags = T

    for i, tag in enumerate(all_tags):
        if ('.', tag) not in transition_prob:
            transition_p = 10**(-7)
        else:
            transition_p = transition_prob[('.', tag)]

        if (observation[0], tag) not in emission_counts:
            emission_p = 10**(-7)
        else:
             emission_p = emission_prob[(observation[0], tag)]
        if emission_p == 0:
            viterbi[i][0] = 0
        else:
            viterbi[i][0] = transition_p * emission_p
        backpointer[i][0] = -1


    for word_index in range(1, t):
        for i, tag in enumerate(all_tags):
            current_max = 0
            arg_max = 8
            for j, prev_tag in enumerate(all_tags):
                if (prev_tag, tag) not in transition_prob:
                    transition_p = 10**(-7)
                else:
                    transition_p = transition_prob[(prev_tag, tag)]
                if (observation[word_index], tag) not in emission_counts:
                    emission_p = 10**(-7)
                else:
                    emission_p = emission_prob[(observation[word_index], tag)]

                if emission_p == 0:
                    most_prob = 0
                else:
                    most_prob = viterbi[j][word_index-1] * transition_p * emission_p

                if most_prob > current_max:
                    current_max = most_prob
                    arg_max = j

            viterbi[i][word_index] = current_max
            backpointer[i][word_index] = arg_max

    current_max = 0
    arg_max = -1
    word_index = t - 1

    for i, prev_tag in enumerate(all_tags):
        if viterbi[i][word_index] > current_max:
            current_max = viterbi[i][word_index]
            arg_max = i

    best_path_prob = current_max
    best_path_pointer = arg_max

    labels = []
    index = best_path_pointer

    for i in range(t-1, -1, -1):
        labels.append(all_tags[index])
        index = backpointer[index][i]
    labels = labels[::-1]

    result = []
    for token in zip(observation, labels):
        result.append(token)

    return result
```

Calculating accuracy on dev dataset using Viterbi algorithm

dev_set - contains all sentences present in dev file. We iterate over each sentence and compute the result. This is appended to result list. Later calculate accuracy for each sentence and find average accurancy over entire dev dataset

In [186]:
```
result = []

# find result for each sentence
for sent in dev_set:
    dev_set_words = []
    for tup in sent:
        dev_set_words.append(tup[0])
    answer = viterbi_algorithm(dev_set_words)
    result.append(answer)

# find accuracy for each sentence
accuracy = []
for num, sent in enumerate(dev_set):

    check = [i for i, j in zip(result[num], sent) if i == j]

    accuracy.append(len(check)/len(result[num]))


# find average accuracy
s = 0
for i in accuracy:
    s += i

print('Viterbi algorithm Accuracy: ',(s/len(accuracy))*100)
```

```
Viterbi algorithm Accuracy:  94.49416882840033
```

In [187]:
```
# find result for each sentence on test set
test_result = []

for sent in test_set:
    answer = viterbi_algorithm(sent)
    test_result.append(answer)
```

Calculating tags using viterbi algorithm on test dataset and writing it to viterbi.out

In [188]:
```
j = 1

url_viterbi = os.path.join(directory, "viterbi.out")

with open(url_viterbi, 'a') as f:
    for sent in test_result:
        for tup in sent:
            f.write(str(j) + "\t" + tup[0] + "\t" + tup[1] + "\n")
            j += 1
        f.write("\n")
```

Accuracy of Viterbi algorithm on Dev set : 94.494 %

In [ ]:

In [ ]:

In [ ]: