Artificial and Computational Intelligence (Assignment - 2)

Problem Statement

As part of the 2nd Assignment, we'll try and predict the Part of Speech (POS) tag for each word in a provided sentence.

You are required to build a model using Hidden Markov Models which would help you predict the POS tags for all words in an utterance.

What is a POS tag?

In corpus linguistics, part-of-speech tagging (POS tagging or PoS tagging or POST), also called grammatical tagging or word-category disambiguation, is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech, based on both its definition and its context—i.e., its relationship with adjacent and related words in a phrase, sentence, or paragraph. A simplified form of this is commonly taught to school-age children, in the identification of words as nouns, verbs, adjectives, adverbs, etc.

Dataset

The dataset can be downloaded from https://drive.google.com/open?id=1345iaxqTImJN6mKGh_c1T5n2OWumpyYz (https://drive.google.com/open?id=1345iaxqTImJN6mKGh_c1T5n2OWumpyYz). You can access it only using your BITS IDs.

Dataset Description

Sample Tuple

b100-5507

Mr. NOUN
Podger NOUN
had VERB
thanked VERB
him PRON
gravely ADV
, .
and CONJ
now ADV
he PRON
made VERB
use NOUN
of ADP
the DET
advice NOUN
. .

Explanation

The first token "b100-5507" is just a key and acts like an identifier to indicate the beginning of a sentence. The other tokens have a (Word, POS Tag) pairing.

List of POS Tags are: .
ADJ
ADP
ADV
CONJ
DET
NOUN
NUM
PRON
PRT
VERB
X

Note
. is used to indicate special characters such as '.', ','
X is used to indicate vocab not part of Enlish Language mostly. Others are Standard POS tags.

Evaluation

We wish to evaluate based on

- coding practices being followed
- commenting to explain the code and logic behind doing something
- your understanding and explanation of data
- how good the model would perform on unseen data.

Train-Test Split

Let us use a 80-20 split of our data for training and evaluation purpose.

In [1]:

```
#Import libraries
import random
from collections import Counter, defaultdict, namedtuple, OrderedDict
from itertools import chain
from pomegranate import State, HiddenMarkovModel, DiscreteDistribution
```

In [7]:

```
#Read data

Sentence = namedtuple("Sentence", "words tags")

# Function to read the data file and tokeizes into sentences .
def read_data(filename):
    """Read tagged sentence data"""
    with open(filename, 'r') as f:
        sentence_lines = [l.split("\n") for l in f.read().split("\n\n")]
        sentences =  OrderedDict(((s[0], Sentence(*zip(*[l.strip().split("\t")
                                for l in s[1:]]))) for s in sentence_lines if s[0]))
    return sentences,sentence_lines
sentences,data=read_data('data.txt')

# Store all the predefined tags. Below logic can be moved into file to make it
# since tag is not stored in file we are just statically storing it .

tags = ['ADJ','ADP','ADV','CONJ','DET','NOUN','NUM','PRON','PRT','VERB','X']
tagset = frozenset(tags)
sentences
```

```
NJ', 'VERB', 'NOUN', 'DET', 'NOUN', 'ADP', 'NOUN', 'CONJ', 'DET', 'NO
UN', 'ADP', 'NOUN', '.'))),
             ('b100-54920',
              Sentence(words=('Could', 'it', 'just', 'be', ',', 'Ther
esa', 'wondered', ',', 'that', 'Anne', 'had', 'understood', 'only',
'too', 'well', ',', 'and', 'that', 'George', 'all', 'along', 'was',
'extraordinary', 'only', 'in', 'the', 'degree', 'to', 'which', 'he',
'was', 'dull', '?', '?'), tags=('VERB', 'PRON', 'ADV', 'VERB', '.',
'NOUN', 'VERB', '.', 'ADP', 'NOUN', 'VERB', 'VERB', 'ADV', 'ADV', 'AD
V', '.', 'CONJ', 'ADP', 'NOUN', 'PRT', 'ADV', 'VERB', 'ADJ', 'ADV',
'ADP', 'DET', 'NOUN', 'ADP', 'DET', 'PRON', 'VERB', 'ADJ', '.',
'.'))),
             ('b100-32087',
              Sentence(words=('Since', 'Af', 'and', 'P', 'divides',
'Af', 'for', 'Af', ',', 'we', 'have', 'Af', '.'), tags=('ADP', 'NOU
N', 'CONJ', 'NOUN', 'VERB', 'NOUN', 'ADP', 'NOUN', '.', 'PRON', 'VER
B', 'NOUN', '.'))),
             ('b100-30402', Sentence(words=('1', '.'), tags=('NUM',
'.'))),
             ('b100-17866'
```

In [3]:

```
#Pre-process data (Whatever you feel might be required)

''' 3. UNIQUE KEY FOR EACH SENTENCE '''
keys = tuple(sentences.keys())

''' 4. CONSTRUCT A VOCABULARY SET OF ALL UNIQUE WORDS '''
wordset = frozenset(chain(*[s.words for s in sentences.values()]))

''' 5. WORD AND TAG SEQUENCES AS A TUPLE SET OF TUPLES '''
word_sequences = tuple([sentences[key].words for key in keys])
tag_sequences = tuple([sentences[key].tags for key in keys])

# written clas and function to split the training and test data which eventuall
# it also has more frequent use logic like tag_Sequcning ,word_Sequencing, word

class Subset(namedtuple("BaseSet", "sentences keys vocab X tagset Y")):

    def __new__(cls, sentences, keys):

        word_sequences = tuple([sentences[key].words for key in keys])
        tag_sequences = tuple([sentences[key].tags for key in keys])
        wordset = frozenset(chain(*word_sequences))
        tagset = frozenset(chain(*tag_sequences))

        return super().__new__(cls,
                               {key: sentences[key] for key in keys},
                               keys,
                               wordset,
                               word_sequences,
                               tagset,
                               tag_sequences)

    def __len__(self):
        return len(self.sentences)

    def __iter__(self):
        return iter(self.sentences.items())

''' 7. SPLIT DATA INTO TRAINING & TEST SETS  Training : Test = 0.8:0.2'''
key_list = list(keys)
train_ratio=0.8
random.shuffle(key_list)
split = int(train_ratio * len(key_list))
training_data = Subset(sentences, key_list[:split])
test_data = Subset(sentences, key_list[split:])

print("Training set has {} sentences.".format(len(training_data.keys)))
print("Test set has {} sentences.\n".format(len(test_data.keys)))
```

```
Training set has 45872 sentences.
Test set has 11468 sentences.
```

In [4]:

```
#Data Description

################ 2. COUNT NUMBER OF EACH TAG IN ENTIRE CORPUS ################
########################  single_tag_counts[tag] = k  ########################
all_sentences = [list(sentence) for sentence in training_data.Y]
all_tags = chain.from_iterable(all_sentences)
single_tag_counts = dict(Counter(all_tags))

############## 3. COUNT NUMBER OF EACH TAG PAIR IN ENTIRE CORPUS ##############
####################  pair_tag_counts[(tag_1, tag_2)] = k  ####################
sentences = [s for s in all_sentences if len(s) > 1]  # discard any sequences o
pairs = []
for s in sentences:
    pairs.extend([(s[i-1], s[i]) for i in range(1, len(s))])

pair_tag_counts = dict(Counter(pairs))

if len(pair_tag_counts) < len(training_data.tagset)**2:
    for tag1 in training_data.tagset:
        for tag2 in training_data.tagset:
            if (tag1, tag2) not in pair_tag_counts:
                pair_tag_counts[(tag1, tag2)] = 0

## 4. COUNT NUMBER OF EACH TAG APPEARING IN THE BEGINNING or END OF SENTENCE ##
#########################  start_tag_counts[tag] = k  #########################
##########################  end_tag_counts[tag] = k  ##########################

start_tag_counts = dict(Counter([sentence[0] for sentence in training_data.Y]))
end_tag_counts = dict(Counter([sentence[-1] for sentence in training_data.Y]))

### if any tag has NO sentences starting/ending with it, set its value to 0:
if len(start_tag_counts) < len(training_data.tagset):
    for tag in training_data.tagset:
        if tag not in start_tag_counts:
            start_tag_counts[tag] = 0

if len(end_tag_counts) < len(training_data.tagset):
    for tag in training_data.tagset:
        if tag not in end_tag_counts:
            end_tag_counts[tag] = 0

################# 5. COUNT NUMBER OF (TAG_i, WORD_i) PAIRS #################
######################  pair_counts[tag][word] = k  ########################
pair_counts = defaultdict(lambda: defaultdict(lambda : 0))

for sentence_idx, sentence in enumerate(training_data.Y):
    for word_idx, tag in enumerate(sentence):
        word = training_data.X[sentence_idx][word_idx]
        pair_counts[tag][word] += 1

```

In [5]:

```
#HMM Model Goes Here

############################# 6. BUILD HMM MODEL ##############################
HMM_model = HiddenMarkovModel(name = "HMM-Tagger")
tag_states = []  # state for each tag

################# (6.1) ADD STATES w/ EMISSION PROBABILITIES #################

for tag in training_data.tagset:
    tag_emissions = DiscreteDistribution({word:pair_counts[tag][word]/single_ta
                                          for word in training_data.vocab})
    tag_state = State(tag_emissions, name = tag)
    tag_states.append(tag_state)
    HMM_model.add_states(tag_state)

############## (6.2) ADD TRANSITIONS w/ TRANSITION PROBABILITIES #############

n_sentences = len(training_data.keys)

for tag_state1 in tag_states:
    for tag_state2 in tag_states:
        tag1, tag2 = tag_state1.name, tag_state2.name
        HMM_model.add_transition(HMM_model.start, tag_state1, start_tag_counts[
        HMM_model.add_transition(tag_state1, HMM_model.end, end_tag_counts[tag1
        HMM_model.add_transition(tag_state1, tag_state2, pair_tag_counts[(tag1,

HMM_model.bake()
```

In [6]:

```
#Model Accuracy Evaluation

################### 7. MAKE PREDICTIONS ON TRAININING SET ###################
##################### DISPLAY THE OUTPUT ON CONSOLE ##########################

train_correct = 0 # number of correct predictions so far
train_count = 0   # number of predictions so far
print_i = 100

### ITERATE PER SENTENCE
for words, true_tags in zip(training_data.X, training_data.Y):
    try:
        # Viterbi Path: most likely sequence of STATES that generated the seque
        _, viterbi_path = HMM_model.viterbi([w for w in words])
        predicted_tags = [state[1].name for state in viterbi_path[1:-1]]
        train_correct += sum(pred == true for pred, true in zip(predicted_tags,

        if print_i == 100: # print a sample result
            print("Training Sentence: \n", words)
            print()
            print("Predicted Tags: \n", predicted_tags)
            print()
            print("True Tags: \n", true_tags)
            print_i += 1
    except:
        pass
    train_count += len(words)

train_acc = train_correct/train_count
print("\nTraining Accuracy: {:.2f}%".format(100 * train_acc))
print()
print()
```

```
Training Sentence:
 ('And', 'Early', 'Spring', 'seized', 'the', 'whip', 'and', 'said',
':')

Predicted Tags:
 ['CONJ', 'ADJ', 'NOUN', 'VERB', 'DET', 'NOUN', 'CONJ', 'VERB', '.']

True Tags:
 ('CONJ', 'ADJ', 'NOUN', 'VERB', 'DET', 'NOUN', 'CONJ', 'VERB', '.')

Training Accuracy: 97.55%
```

In [110]:

```
#Adds code blocks wherever you feel necessary

####################### 8. MAKE PREDICTIONS ON TEST SET ######################
###################### DISPLAY THE OUTPUT ON CONSOLE ##########################
test_correct = 0
test_count = 0

### ITERATE PER SENTENCE
for words, true_tags in zip(test_data.X, test_data.Y):
    try:
        # Only consider words contained in training set's vocab
        _, viterbi_path = HMM_model.viterbi([w if w in training_data.vocab else
        predicted_tags = [state[1].name for state in viterbi_path[1:-1]]
        test_correct += sum(pred == true for pred, true in zip(predicted_tags, t

        if print_i == 101: # print a sample result
            print("Test Sentence: \n", words)
            print()
            print("Predicted Tags: \n", predicted_tags)
            print()
            print("True Tags: \n", true_tags)
            print_i += 1
    except:
        pass
    test_count += len(words)

test_acc = test_correct/test_count
print("\nTest Accuracy: {:.2f}%".format(100 * test_acc))
```

```
Test Sentence:
 ('Police', 'said', 'he', 'became', 'ill', 'while', 'parked', 'in', 'f
ront', 'of', 'a', 'barber', 'shop', 'at', '229', 'West', 'Pratt', 'Str
eet', '.')

Predicted Tags:
 ['NOUN', 'VERB', 'PRON', 'VERB', 'ADJ', 'NOUN', 'VERB', 'ADP', 'NOU
N', 'ADP', 'DET', 'NOUN', 'NOUN', 'ADP', 'NUM', 'ADJ', 'NOUN', 'NOUN',
'.']

True Tags:
 ('NOUN', 'VERB', 'PRON', 'VERB', 'ADJ', 'ADP', 'VERB', 'ADP', 'NOUN',
'ADP', 'DET', 'NOUN', 'NOUN', 'ADP', 'NUM', 'ADJ', 'NOUN', 'NOUN',
'.')

Test Accuracy: 95.98%
```

Happy Coding!