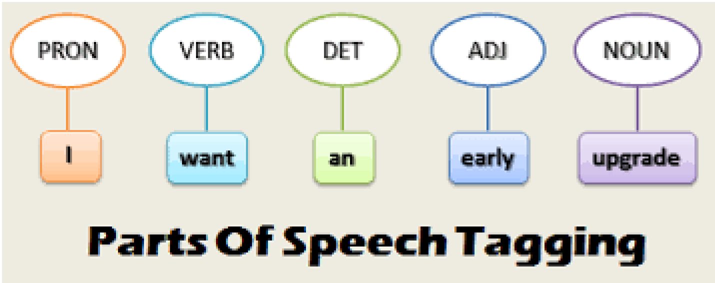
Implementing HMM for POS Tagging

In this notebook we will implement a Hidden Markov Model for Parts-of-Speech Tagging.

Associating each word in a sentence with a proper POS (part of speech) is known as POS tagging or POS annotation. POS tags are also known as word classes, morphological classes, or lexical tags. The tag in case of is a POS tag, and signifies whether the word is a noun, adjective, verb, and so on.



In [1]:

```
import nltk
import numpy as np

from tqdm import tqdm
```

In [2]:

```
# Inorder to get the notebooks running in current directory
import os, sys, inspect
currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.currentframe()))
parentdir = os.path.dirname(currentdir)
sys.path.insert(0, parentdir)

import hmm
```

We will be making use of the Treebank corpora with the Universal Tagset.

The Treebank corpora provide a syntactic parse for each sentence. The NLTK data package includes a 10% sample of the Penn Treebank (in treebank), as well as the Sinica Treebank (in sinica_treebank).

Not all corpora employ the same set of tags. Initially we want to avoid the complications of these tagsets, so we use a built-in mapping to the "Universal Tagset".

In [3]:

```
# Download the treebank corpus from nltk
nltk.download('treebank')

# Download the universal tagset from nltk
nltk.download('universal_tagset')
```

```
[nltk_data] Downloading package treebank to
[nltk_data]     /Users/kad99kev/nltk_data...
[nltk_data]   Package treebank is already up-to-date!
[nltk_data] Downloading package universal_tagset to
[nltk_data]     /Users/kad99kev/nltk_data...
[nltk_data]   Package universal_tagset is already up-to-date!
```

Out[3]:

```
True
```

In [4]:

```
# Reading the Treebank tagged sentences
nltk_data = list(nltk.corpus.treebank.tagged_sents(tagset='universal'))
```

A Look At Our Data

Let's take a look at the data we have

We have a total of *100,676* words tagged.

This includes a total of *12* unique tags with *12,408* unique words.

In [5]:

```
# Sample Output
for (word, tag) in nltk_data[0]:
    print(f"Word: {word} | Tag: {tag}")
```

```
Word: Pierre | Tag: NOUN
Word: Vinken | Tag: NOUN
Word: , | Tag: .
Word: 61 | Tag: NUM
Word: years | Tag: NOUN
Word: old | Tag: ADJ
Word: , | Tag: .
Word: will | Tag: VERB
Word: join | Tag: VERB
Word: the | Tag: DET
Word: board | Tag: NOUN
Word: as | Tag: ADP
Word: a | Tag: DET
Word: nonexecutive | Tag: ADJ
Word: director | Tag: NOUN
Word: Nov. | Tag: NOUN
Word: 29 | Tag: NUM
Word: . | Tag: .
```

In [6]:

```
tagged_words = [tags for sent in nltk_data for tags in sent]
```

In [7]:

```
print(f"Size of tagged words: {len(tagged_words)}")
print(f"Example: {tagged_words[0]}")
```

```
Size of tagged words: 100676
Example: ('Pierre', 'NOUN')
```

In [8]:

```
tags = list({tag for (word, tag) in tagged_words})
print(f"Tags: {tags} | Number of tags: {len(tags)}")
```

```
Tags: ['CONJ', '.', 'PRT', 'NUM', 'PRON', 'ADV', 'DET', 'ADJ', 'ADP',
'X', 'VERB', 'NOUN'] | Number of tags: 12
```

In [9]:

```
words = list({word for (word, tag) in tagged_words})
print(f"First 15 Words: {words[:15]} | Number of words: {len(words)}")
```

```
First 15 Words: ['dispute', "O'Loughlin", 'motion', 'posted', 'erudit
e', 'cow', '321', 'seeks', 'replicated', 'kinds', 'comparable', 'Repub
lican', 'merchandise', '326', 'nearby'] | Number of words: 12408
```

Computing Transition and Emission Matrices

Once we have our data ready, we will need to create our transition and emission matrices.

Inorder to do this, we need to understand how we calculate these probability matrices.

For Transition Matrices

- For a given source_tag and destination_tag do:
  - Get total counts of source_tag in corpus (all_tags)
  - Loop through all_tags and do:
    - Get all counts of instances where at timestep i, the source_tag had dest_tag at timestep i + 1
  - Get probability for dest_tag given source_tag as *P(destintation tag | source tag) = Count of destination tag to source tag / Count of source tag*

For Emission Matrices

- For a given word and tag do:
  - Get a list of (word, tag) from each pair of tagged words such that the iterating tag matches the given tag.
  - From this stored tags that was created from the given tag, create a list of words for which the iterating word matches the given word
  - Using the counts of the word given a tag and the total occurances of a tag, we compute the conditional probability *P(word | tag) = Count of word and tag / Count of given tag*

In [10]:

```
def compute_transition_matrix(tags, tagged_words):

    all_tags = [tag for (_, tag) in tagged_words]

    def compute_counts(dest_tag, source_tag):
        count_source = len([t for t in all_tags if t == source_tag])
        count_dest_source = 0
        for i in range(len(all_tags) - 1):
            if all_tags[i] == source_tag and all_tags[i + 1] == dest_tag:
                count_dest_source += 1
        return count_dest_source, count_source

    trans_matrix = np.zeros((len(tags), len(tags)))

    for i, source_tag in enumerate(tags):
        for j, dest_tag in enumerate(tags):
            count_dest_source, count_source = compute_counts(dest_tag, source_tag)
            trans_matrix[i, j] = count_dest_source / count_source

    return trans_matrix
```

In [11]:

```
# transition_matrix = compute_transition_matrix(tags, tagged_words)
```

In [12]:

```
# Computing Emission Probability
def compute_emission_matrix(words, tags, tagged_words):

    def compute_counts(given_word, given_tag):
        tags = [word_tag for word_tag in tagged_words if word_tag[1] == given_tag]
        word_given_tag = [word for (word, _) in tags if word == given_word]
        return len(word_given_tag), len(tags)

    emi_matrix = np.zeros((len(tags), len(words)))

    for i, tag in enumerate(tags):
        for j, word in enumerate(tqdm(words, desc=f"Current Tag - {tag}")):
            count_word_given_tag, count_tag = compute_counts(word, tag)
            emi_matrix[i, j] = count_word_given_tag / count_tag

    return emi_matrix
```

In [13]:

```
# emission_matrix = compute_emission_matrix(words, tags, tagged_words)
```

In [14]:

```
def save_matrices(observable_states, emission_matrix, hidden_states, transition_matr
    try:
        os.mkdir(save_dir)
    except FileExistsError:
        raise FileExistsError(
            "Directory already exists! Please provide a different output directory!'
        )

    np.save(save_dir + '/observable_states', observable_states)
    np.save(save_dir + '/emission_matrix', emission_matrix)
    np.save(save_dir + '/hidden_states', hidden_states)
    np.save(save_dir + '/transition_matrix', transition_matrix)
```

In [15]:

```
# save_matrices(words, emission_matrix, tags, transition_matrix)
```

In [16]:

```
def load_matrices(save_dir):
    observable_states = np.load(save_dir + '/observable_states.npy')
    emission_matrix = np.load(save_dir + '/emission_matrix.npy')
    hidden_states = np.load(save_dir + '/hidden_states.npy')
    transition_matrix = np.load(save_dir + '/transition_matrix.npy')
    return observable_states.tolist(), emission_matrix, hidden_states.tolist(), trar
```

In [17]:

```
observable_states, emission_matrix, hidden_states, transition_matrix = load_matrices
```

Let us take a look at some of the observed and hidden states

In [18]:

```
observable_states[:15], hidden_states
```

Out[18]:

```
(['convinced',
  'counting',
  'Anti-Deficiency',
  'unrealized',
  '6.40',
  '382-37',
  'actor',
  'Rouge',
  'Marc',
  'critics',
  'eliminated',
  'Secretary',
  'corners',
  '*T*-81',
  'least'],
 ['DET',
  'ADJ',
  'VERB',
  'PRT',
  'X',
  'PRON',
  'ADV',
  'CONJ',
  'NUM',
  '.',
  'ADP',
  'NOUN'])
```

We will need to write a function that tokenizes the input sentence with the index specified by our saved words list.

In [19]:

```
def tokenize(input_sent, words):
    lookup = {word: i for i, word in enumerate(words)}

    tokenized = []
    input_sent = input_sent.split(' ')
    for word in input_sent:
        idx = lookup[word]
        tokenized.append(idx)

    return tokenized
```

Run The Markov Model

Let us now run our Hidden Markov Model with the observered and hidden states with our transition and emission matrices.

In [20]:

```
model = hmm.HiddenMarkovModel(
    observable_states, hidden_states, transition_matrix, emission_matrix
)
```

In [ ]:

```
model.print_model_info()
```

In [22]:

```
input_sent = 'Pierre Vinken , 61 years old , will join the board as a nonexecutive d
input_tokens = tokenize(input_sent, observable_states)
print(input_tokens)
```

```
[459, 9165, 5817, 483, 3096, 10713, 5817, 9219, 166, 8733, 2232, 1236
7, 9530, 2522, 2367, 1640, 1038, 9079]
```

Forward Algorithm

In [ ]:

```
alpha, a_probs = model.forward(input_tokens)
hmm.print_forward_result(alpha, a_probs)
```

Backward Algorithm

Let us verify the output of the Forward Algorithm by running the Backward Algorithm.

In [ ]:

```
beta, b_probs = model.backward(input_tokens)
hmm.print_backward_result(beta, b_probs)
```

Viterbi Algorithm

This algorithm will give us the POS for each token or word. This is useful to generate POS Tagger for different sentences.

In [ ]:

```
path, delta, phi = model.viterbi(input_tokens)
hmm.print_viterbi_result(input_tokens, observable_states, hidden_states, path, delta
```

Running the cell above, we get:

```
Result:
     Observation BestPath
0         Pierre     NOUN
1         Vinken     NOUN
2              ,        .
3             61      NUM
4          years     NOUN
5            old      ADJ
6              ,        .
7           will     VERB
8           join     VERB
9            the      DET
10         board     NOUN
11            as      ADP
12             a      DET
13  nonexecutive      ADJ
14      director     NOUN
15          Nov.     NOUN
16            29      NUM
17             .        .
```

In [ ]:

```

```