In [ ]:
```
import pandas as pd
import numpy as np
```

In this assignment, you will manually decode the highest-probability sequence of part-of-speech tags from a trained HMM using the Viterbi algorithm. You will also fine-tune BERT-based models for named entity recognition (NER).

1. POS tagging with an HMM

Consider a Hidden Markov Model with the following parameters: postags = {NOUN, AUX, VERB}, words are 'Patrick', 'Cherry', 'can', 'will', 'see', 'spot'

In [ ]:
```
# noun, aux, verb (initial probs \pi)
postags = ['noun', 'aux', 'verb']
postag_probs = [0.7, 0.1, 0.2]

# transition probabilities (P(column tag | row tag))
# noun, aux, verb, aux | noun = 0.3
transitions = [[0.2, 0.3, 0.5],
    [0.4, 0.1, 0.5],
    [0.8, 0.1, 0.1]]

# patrick, cherry, can, will, see, spot
obs = ["Patrick", 'Cherry', 'can','will','see','spot']
emissions = [[0.3, 0.2, 0.1, 0.1, 0.1, 0.2],
             [0, 0, 0.4, 0.6, 0, 0],
             [0, 0, 0.1, 0.2, 0.5, 0.2]]
transitions = np.array(transitions)
emissions = np.array(emissions)
col_index = {i: word for i, word in enumerate(obs)}
reverse_col_index = {word: i for i, word in col_index.items()}
row_index = {i: pos for i, pos in enumerate(postags)}
reverse_row_index = {word: i for i, word in row_index.items()}
```

In [ ]:
```
sentences = ["Patrick can see Cherry", "will Cherry spot Patrick"]
```

In [ ]:
```
# observations of len T, stage-graph of len N) returns best-path, path-prob
def viterbi(observations, state_graph):
    N = len(state_graph)
    T = len(observations)
    vtable, backpointer = np.zeros([N, T]), np.zeros([T])
    # create a path probability matrix viterbi[N,T]
    for state in range(len(postags)):
        vtable[state, 0] = postag_probs[state] * emissions[state,reverse_col_index[observations[0]]]
        backpointer[0] = np.argmax(vtable[state,:])
    for t in range(1, len(observations)):
        max_state_idx = np.argmax(vtable[:,t-1])
        for state in range(len(postags)):
            vtable[state, t] = vtable[max_state_idx, t-1] * transitions[max_state_idx,state] * emiss

        backpointer[t] = np.argmax(vtable[:, t])
    bestpathprob = max(vtable[:,T - 1])
    bestpathpointer = np.argmax(vtable[:, T-1])
    bestpath = np.array(backpointer + [bestpathpointer])

    return vtable, bestpath, bestpathprob
```

In [ ]:
```
vtable1, bp1, bpp1 = viterbi(sentences[0].split(), postags)
vtable2, bp2, bpp2 = viterbi(sentences[1].split(), postags)
print(["Sentence 1 tag seq"] + [postags[word] for word in map(int, bp1)])
print(["Sentence 1 bestpathprob"] + [bpp1])
print("Viterbi Table for Sentence 1:\n", vtable1)
print(["Sentence 2 tag seq"] + [postags[word] for word in map(int, bp2)])
print(["Sentence 2 bestpathprob"] + [bpp2])
print("Viterbi Table for Sentence 2:\n", vtable2)
```

```
['Sentence 1 tag seq', 'noun', 'aux', 'verb', 'noun']
['Sentence 1 bestpathprob', 0.001008]
Viterbi Table for Sentence 1:
 [[0.21     0.0042   0.001008 0.001008]
 [0.       0.0252   0.       0.      ]
 [0.       0.0105   0.0063   0.      ]]
['Sentence 2 tag seq', 'noun', 'noun', 'verb', 'noun']
['Sentence 2 bestpathprob', 6.720000000000001e-05]
Viterbi Table for Sentence 2:
 [[7.00e-02 2.80e-03 1.12e-04 6.72e-05]
 [6.00e-02 0.00e+00 0.00e+00 0.00e+00]
 [4.00e-02 0.00e+00 2.80e-04 0.00e+00]]
```

The way I made my Viterbi tables was by creating a function according to the Viterbi algorithms in the textbook, with some minor modifications, because what they wrote in their algorithm didn't completely concord with what they stated in the text.

The algorithm involves identifying the number of states and the number of words in the sentence. It then initializes the viterbi table (a state-outcome mapping), in this context a POS-Word mapping with the POS in the rows and Words in the columns as we read from left to right.

It then obtains the the conditional probability of emitting the first word in the sentence. Generatively speaking, this is the prior probability over parts of speech times the conditional probability of the word given states.

The algorithm also involves a "backpointer". This backpointer indicates the state with the highest probability up to the current observation/word. As such, the np.argmax function is used to obtain the index of the cell at a given point that is highest probability. In other words, as we read left to right the sentence, the probability of a state contains the information and probabilities of all the previous states. This is thanks to the markov assumption, where a current state is conditionally of all previous states given just the last previous state.

AFter the first word, for the rest of the sentence, we use the last states probability as our new prior probability and the state on which we condition our transitions to new states to obtain the conditional probability of each possible new state. This conditional probability of each new state then links to the emission probability of the current word via the chain rule of probability. In this way, we obtain the joint probability of our previous state, the possible next state give our previous state, and the current observation. We can then use the backpointer to indicate which new state maximizes the probability of our current observation. This can repeat until we finish the sentence.

The algorithm in my case then returns the entire Viterbi table, the best path through the states that maximize the likelihood of the observations in sequence, and the probability of this best path.