C2_W2_Assignment

October 31, 2020

1 Assignment 2: Parts-of-Speech Tagging (POS)

Welcome to the second assignment of Course 2 in the Natural Language Processing specialization. This assignment will develop skills in part-of-speech (POS) tagging, the process of assigning a part-of-speech tag (Noun, Verb, Adjective…) to each word in an input text. Tagging is difficult because some words can represent more than one part of speech at different times. They are **Ambiguous**. Let's look at the following example:

- The whole team played **well**. [adverb]
- You are doing **well** for yourself. [adjective]
- **Well**, this assignment took me forever to complete. [interjection]
- The **well** is dry. [noun]
- Tears were beginning to **well** in her eyes. [verb]

Distinguishing the parts-of-speech of a word in a sentence will help you better understand the meaning of a sentence. This would be critically important in search queries. Identifying the proper noun, the organization, the stock symbol, or anything similar would greatly improve everything ranging from speech recognition to search. By completing this assignment, you will:

- Learn how parts-of-speech tagging works
- Compute the transition matrix A in a Hidden Markov Model
- Compute the emission matrix B in a Hidden Markov Model
- Compute the Viterbi algorithm
- Compute the accuracy of your own model

1.1 Outline

```
In [ ]: # Importing packages and loading in the data set
        from utils_pos import get_word_tag, preprocess
        import pandas as pd
        from collections import defaultdict
        import math
        import numpy as np
```

## Part 0: Data Sources This assignment will use two tagged data sets collected from the **Wall Street Journal (WSJ)**.

Here is an example 'tag-set' or Part of Speech designation describing the two or three letter tag and their meaning. - One data set (**WSJ-2_21.pos**) will be used for **training**. - The other (**WSJ-24.pos**) for **testing**. - The tagged training data has been preprocessed to form a vocabulary (**hmm_vocab.txt**). - The words in the vocabulary are words from the training set that were used two or more times. - The vocabulary is augmented with a set of 'unknown word tokens', described below.

The training set will be used to create the emission, transmission and tag counts.

The test set (WSJ-24.pos) is read in to create `y`. - This contains both the test text and the true tag. - The test set has also been preprocessed to remove the tags to form **test_words.txt**. - This is read in and further processed to identify the end of sentences and handle words not in the vocabulary using functions provided in **utils_pos.py**. - This forms the list `prep`, the preprocessed text used to test our POS taggers.

A POS tagger will necessarily encounter words that are not in its datasets. - To improve accuracy, these words are further analyzed during preprocessing to extract available hints as to their appropriate tag. - For example, the suffix 'ize' is a hint that the word is a verb, as in 'final-ize' or 'character-ize'. - A set of unknown-tokens, such as '–unk-verb–' or '–unk-noun–' will replace the unknown words in both the training and test corpus and will appear in the emission, transmission and tag data structures.

Implementation note:

- For python 3.6 and beyond, dictionaries retain the insertion order.
- Furthermore, their hash-based lookup makes them suitable for rapid membership tests.
    - If *di* is a dictionary, `key in di` will return `True` if *di* has a key *key*, else `False`.

The dictionary vocab will utilize these features.

```
In [ ]: # load in the training corpus
        with open("WSJ_02-21.pos", 'r') as f:
            training_corpus = f.readlines()

        print(f"A few items of the training corpus list")
        print(training_corpus[0:5])
```

```
In [ ]: # read the vocabulary data, split by each line of text, and save the list
        with open("hmm_vocab.txt", 'r') as f:
            voc_l = f.read().split('\n')

        print("A few items of the vocabulary list")
        print(voc_l[0:50])
        print()
        print("A few items at the end of the vocabulary list")
        print(voc_l[-50:])
```

```
In [ ]: # vocab: dictionary that has the index of the corresponding words
        vocab = {}

        # Get the index of the corresponding words.
        for i, word in enumerate(sorted(voc_l)):
            vocab[word] = i

        print("Vocabulary dictionary, key is the word, value is a unique integer")
        cnt = 0
        for k,v in vocab.items():
            print(f"{k}:{v}")
            cnt += 1
            if cnt > 20:
                break
```

```
In [ ]: # load in the test corpus
        with open("WSJ_24.pos", 'r') as f:
            y = f.readlines()

        print("A sample of the test corpus")
        print(y[0:10])
```

```
In [ ]: #corpus without tags, preprocessed
        _, prep = preprocess(vocab, "test.words")

        print('The length of the preprocessed test corpus: ', len(prep))
        print('This is a sample of the test_corpus: ')
        print(prep[0:10])
```

# Part 1: Parts-of-speech tagging

## Part 1.1 - Training You will start with the simplest possible parts-of-speech tagger and we will build up to the state of the art.

In this section, you will find the words that are not ambiguous. - For example, the word `is` is a verb and it is not ambiguous. - In the `WSJ` corpus, 86% of the token are unambiguous (meaning they have only one tag) - About 14% are ambiguous (meaning that they have more than one tag)

Before you start predicting the tags of each word, you will need to compute a few dictionaries that will help you to generate the tables.

Transition counts

- The first dictionary is the `transition_counts` dictionary which computes the number of times each tag happened next to another tag.

This dictionary will be used to compute:

$$P(t_i|t_{i-1}) \tag{1}$$

This is the probability of a tag at position i given the tag at position $i-1$.

In order for you to compute equation 1, you will create a `transition_counts` dictionary where - The keys are `(prev_tag, tag)` - The values are the number of times those two tags appeared in that order.

Emission counts The second dictionary you will compute is the `emission_counts` dictionary. This dictionary will be used to compute:

$$P(w_i|t_i) \tag{2}$$

In other words, you will use it to compute the probability of a word given its tag.

In order for you to compute equation 2, you will create an `emission_counts` dictionary where - The keys are `(tag, word)` - The values are the number of times that pair showed up in your training set.

Tag counts The last dictionary you will compute is the `tag_counts` dictionary. - The key is the tag - The value is the number of times each tag appeared.

### Exercise 01

Instructions: Write a program that takes in the `training_corpus` and returns the three dictionaries mentioned above `transition_counts`, `emission_counts`, and `tag_counts`. - `emission_counts`: maps (tag, word) to the number of times it happened. - `transition_counts`: maps (prev_tag, tag) to the number of times it has appeared. - `tag_counts`: maps (tag) to the number of times it has occured.

Implementation note: This routine utilises *defaultdict*, which is a subclass of *dict*. - A standard Python dictionary throws a *KeyError* if you try to access an item with a key that is not currently in the dictionary. - In contrast, the *defaultdict* will create an item of the type of the argument, in this case an integer with the default value of 0. - See defaultdict.

```
In [ ]: # UNQ_C1 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
        # GRADED FUNCTION: create_dictionaries
        def create_dictionaries(training_corpus, vocab):
            """
```

```
Input:
    training_corpus: a corpus where each line has a word followed by its tag.
    vocab: a dictionary where keys are words in vocabulary and value is an index
Output:
    emission_counts: a dictionary where the keys are (tag, word) and the values ar
    transition_counts: a dictionary where the keys are (prev_tag, tag) and the val
    tag_counts: a dictionary where the keys are the tags and the values are the co
"""

# initialize the dictionaries using defaultdict
emission_counts = defaultdict(int)
transition_counts = defaultdict(int)
tag_counts = defaultdict(int)

# Initialize "prev_tag" (previous tag) with the start state, denoted by '--s--'
prev_tag = '--s--'

# use 'i' to track the line number in the corpus
i = 0

# Each item in the training corpus contains a word and its POS tag
# Go through each word and its tag in the training corpus
for word_tag in training_corpus:

    # Increment the word_tag count
    i += 1

    # Every 50,000 words, print the word count
    if i % 50000 == 0:
        print(f"word count = {i}")

    ### START CODE HERE (Replace instances of 'None' with your code) ###
    # get the word and tag using the get_word_tag helper function (imported from u
    word, tag = get_word_tag(word_tag,vocab)

    # Increment the transition count for the previous word and tag
    transition_counts[(prev_tag, tag)] += 1

    # Increment the emission count for the tag and word
    emission_counts[(tag, word)] += 1

    # Increment the tag count
    tag_counts[tag] += 1

    # Set the previous tag to this tag (for the next iteration of the loop)
    prev_tag = tag

    ### END CODE HERE ###
```

```
        return emission_counts, transition_counts, tag_counts
```

```
In [ ]: emission_counts, transition_counts, tag_counts = create_dictionaries(training_corpus, 
```

```
In [ ]: # get all the POS states
        states = sorted(tag_counts.keys())
        print(f"Number of POS tags (number of 'states'): {len(states)}")
        print("View these POS tags (states)")
        print(states)
```

Expected Output

```
Number of POS tags (number of 'states'46
View these states
['#', '$', "''", '(', ')', ',', '--s--', '.', ':', 'CC', 'CD', 'DT', 'EX', 'FW', 'IN', 'JJ', '
```

The ‘states’ are the Parts-of-speech designations found in the training data. They will also be referred to as ‘tags’ or POS in this assignment.

- “NN” is noun, singular,
- ‘NNS’ is noun, plural.
- In addition, there are helpful tags like ‘–s–’ which indicate a start of a sentence.
- You can get a more complete description at Penn Treebank II tag set.

```
In [ ]: print("transition examples: ")
        for ex in list(transition_counts.items())[:3]:
            print(ex)
        print()

        print("emission examples: ")
        for ex in list(emission_counts.items())[200:203]:
            print (ex)
        print()

        print("ambiguous word example: ")
        for tup,cnt in emission_counts.items():
            if tup[1] == 'back': print (tup, cnt)
```

Expected Output

```
transition examples:
(('--s--', 'IN'), 5050)
(('IN', 'DT'), 32364)
(('DT', 'NNP'), 9044)

emission examples:
(('DT', 'any'), 721)
(('NN', 'decrease'), 7)
```

```
(('NN', 'insider-trading'), 5)

ambiguous word example:
('RB', 'back') 304
('VB', 'back') 20
('RP', 'back') 84
('JJ', 'back') 25
('NN', 'back') 29
('VBP', 'back') 4
```

### Part 1.2 - Testing

Now you will test the accuracy of your parts-of-speech tagger using your `emission_counts` dictionary. - Given your preprocessed test corpus `prep`, you will assign a parts-of-speech tag to every word in that corpus. - Using the original tagged test corpus `y`, you will then compute what percent of the tags you got correct.

### Exercise 02

Instructions: Implement `predict_pos` that computes the accuracy of your model.

- This is a warm up exercise.
- To assign a part of speech to a word, assign the most frequent POS for that word in the training set.
- Then evaluate how well this approach works. Each time you predict based on the most frequent POS for the given word, check whether the actual POS of that word is the same. If so, the prediction was correct!
- Calculate the accuracy as the number of correct predictions divided by the total number of words for which you predicted the POS tag.

```
In [ ]: # UNQ_C2 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
        # GRADED FUNCTION: predict_pos

        def predict_pos(prep, y, emission_counts, vocab, states):
            '''
            Input:
                prep: a preprocessed version of 'y'. A list with the 'word' component of the t
                y: a corpus composed of a list of tuples where each tuple consists of (word, P
                emission_counts: a dictionary where the keys are (tag,word) tuples and the val
                vocab: a dictionary where keys are words in vocabulary and value is an index
                states: a sorted list of all possible tags for this assignment
            Output:
                accuracy: Number of times you classified a word correctly
            '''

            # Initialize the number of correct predictions to zero
            num_correct = 0

            # Get the (tag, word) tuples, stored as a set
            all_words = set(emission_counts.keys())

            # Get the number of (word, POS) tuples in the corpus 'y'
```

```
total = len(y)
for word, y_tup in zip(prep, y):

    # Split the (word, POS) string into a list of two items
    y_tup_l = y_tup.split()

    # Verify that y_tup contain both word and POS
    if len(y_tup_l) == 2:

        # Set the true POS label for this word
        true_label = y_tup_l[1]

    else:
        # If the y_tup didn't contain word and POS, go to next word
        continue

    count_final = 0
    pos_final = ''

    # If the word is in the vocabulary...
    if word in vocab:
        for pos in states:

        ### START CODE HERE (Replace instances of 'None' with your code) ###

            # define the key as the tuple containing the POS and word
            key = (pos,word)

            # check if the (pos, word) key exists in the emission_counts dictionar
            if key in emission_counts: # complete this line

            # get the emission count of the (pos,word) tuple
                count = emission_counts[key]

                # keep track of the POS with the largest count
                if count>count_final: # complete this line

                    # update the final count (largest count)
                    count_final = count

                    # update the final POS
                    pos_final = pos

        # If the final POS (with the largest count) matches the true POS:
        if pos_final == true_label: # complete this line

            # Update the number of correct predictions
            num_correct += 1
### END CODE HERE ###
accuracy = num_correct / total

return accuracy
```

```
In [ ]: accuracy_predict_pos = predict_pos(prep, y, emission_counts, vocab, states)
        print(f"Accuracy of prediction using predict_pos is {accuracy_predict_pos:.4f}")
```

Expected Output

```
Accuracy of prediction using predict_pos is 0.8889
```

88.9% is really good for this warm up exercise. With hidden markov models, you should be able to get **95% accuracy.**

# Part 2: Hidden Markov Models for POS

Now you will build something more context specific. Concretely, you will be implementing a Hidden Markov Model (HMM) with a Viterbi decoder - The HMM is one of the most commonly used algorithms in Natural Language Processing, and is a foundation to many deep learning techniques you will see in this specialization. - In addition to parts-of-speech tagging, HMM is used in speech recognition, speech synthesis, etc. - By completing this part of the assignment you will get a 95% accuracy on the same dataset you used in Part 1.

The Markov Model contains a number of states and the probability of transition between those states. - In this case, the states are the parts-of-speech. - A Markov Model utilizes a transition matrix, `A`. - A Hidden Markov Model adds an observation or emission matrix `B` which describes the probability of a visible observation when we are in a particular state. - In this case, the emissions are the words in the corpus - The state, which is hidden, is the POS tag of that word.

## Part 2.1 Generating Matrices

1.1.1 Creating the 'A' transition probabilities matrix

Now that you have your `emission_counts`, `transition_counts`, and `tag_counts`, you will start implementing the Hidden Markov Model.

This will allow you to quickly construct the - `A` transition probabilities matrix. - and the `B` emission probabilities matrix.

You will also use some smoothing when computing these matrices.

Here is an example of what the `A` transition matrix would look like (it is simplified to 5 tags for viewing. It is 46x46 in this assignment.):

A	...	RBS	RP	SYM	TO	UH	...
RBS	...	2.217069e-06	2.217069e-06	2.217069e-06	0.008870	2.217069e-06	...
RP	...	3.756509e-07	7.516775e-04	3.756509e-07	0.051089	3.756509e-07	...
SYM	...	1.722772e-05	1.722772e-05	1.722772e-05	0.000017	1.722772e-05	...
TO	...	4.477336e-05	4.472863e-08	4.472863e-08	0.000090	4.477336e-05	...
UH	...	1.030439e-05	1.030439e-05	1.030439e-05	0.061837	3.092348e-02	...
...	...	...	...	...	...	...	...

Note that the matrix above was computed with smoothing.

Each cell gives you the probability to go from one part of speech to another. - In other words, there is a 4.47e-8 chance of going from parts-of-speech `TO` to `RP`. - The sum of each row has to equal 1, because we assume that the next POS tag must be one of the available columns in the table.

The smoothing was done as follows:

$$P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i) + \alpha}{C(t_{i-1}) + \alpha * N} \tag{3}$$

- N is the total number of tags
- $C(t_{i-1}, t_i)$ is the count of the tuple (previous POS, current POS) in `transition_counts` dictionary.
- $C(t_{i-1})$ is the count of the previous POS in the `tag_counts` dictionary.
- α is a smoothing parameter.

### Exercise 03

Instructions: Implement the `create_transition_matrix` below for all tags. Your task is to output a matrix that computes equation 3 for each cell in matrix `A`.

```
In [ ]: # UNQ_C3 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
        # GRADED FUNCTION: create_transition_matrix
        def create_transition_matrix(alpha, tag_counts, transition_counts):
            '''
            Input:
                alpha: number used for smoothing
                tag_counts: a dictionary mapping each tag to its respective count
                transition_counts: transition count for the previous word and tag
            Output:
                A: matrix of dimension (num_tags,num_tags)
            '''
            # Get a sorted list of unique POS tags
            all_tags = sorted(tag_counts.keys())

            # Count the number of unique POS tags
            num_tags = len(all_tags)

            # Initialize the transition matrix 'A'
            A = np.zeros((num_tags,num_tags))

            # Get the unique transition tuples (previous POS, current POS)
            trans_keys = set(transition_counts.keys())

            ### START CODE HERE (Return instances of 'None' with your code) ###

            # Go through each row of the transition matrix A
            for i in range(num_tags):

                # Go through each column of the transition matrix A
                for j in range(num_tags):

                    # Initialize the count of the (prev POS, current POS) to zero
                    count = 0
                    # Define the tuple (prev POS, current POS)
                    # Get the tag at position i and tag at position j (from the all_tags list)
                    key = (all_tags[i],all_tags[j])

                    # Check if the (prev POS, current POS) tuple
                    # exists in the transition counts dictionaory
                    if transition_counts: #complete this line

                        # Get count from the transition_counts dictionary
                        # for the (prev POS, current POS) tuple
                        count = transition_counts[key]

                    # Get the count of the previous tag (index position i) from tag_counts
                    count_prev_tag = tag_counts[all_tags[i]]

                    # Apply smoothing using count of the tuple, alpha,
                    # count of previous tag, alpha, and number of total tags
                    A[i,j] = (count + alpha) / (count_prev_tag + alpha*num_tags)

            ### END CODE HERE ###

            return A
```

```
In [ ]: alpha = 0.001
        A = create_transition_matrix(alpha, tag_counts, transition_counts)
        # Testing your function
        print(f"A at row 0, col 0: {A[0,0]:.9f}")
        print(f"A at row 3, col 1: {A[3,1]:.4f}")

        print("View a subset of transition matrix A")
        A_sub = pd.DataFrame(A[30:35,30:35], index=states[30:35], columns = states[30:35] )
        print(A_sub)
```

Expected Output

```
A at row 0, col 0: 0.000007040
A at row 3, col 1: 0.1691
View a subset of transition matrix A
              RBS            RP           SYM        TO            UH
RBS  2.217069e-06  2.217069e-06  2.217069e-06  0.008870  2.217069e-06
RP   3.756509e-07  7.516775e-04  3.756509e-07  0.051089  3.756509e-07
SYM  1.722772e-05  1.722772e-05  1.722772e-05  0.000017  1.722772e-05
TO   4.477336e-05  4.472863e-08  4.472863e-08  0.000090  4.477336e-05
UH   1.030439e-05  1.030439e-05  1.030439e-05  0.061837  3.092348e-02
```

1.1.2 Create the 'B' emission probabilities matrix

Now you will create the `B` transition matrix which computes the emission probability.

You will use smoothing as defined below:

$$P(w_i|t_i) = \frac{C(t_i, word_i) + \alpha}{C(t_i) + \alpha * N} \tag{4}$$

- $C(t_i, word_i)$ is the number of times $word_i$ was associated with tag_i in the training data (stored in `emission_counts` dictionary).
- $C(t_i)$ is the number of times tag_i was in the training data (stored in `tag_counts` dictionary).
- N is the number of words in the vocabulary
- α is a smoothing parameter.

The matrix `B` is of dimension (num_tags, N), where num_tags is the number of possible parts-of-speech tags.

Here is an example of the matrix, only a subset of tags and words are shown:

B Emissions Probability Matrix (subset)

B	...	725	adroitly	engineers	promoted	synergy	...
CD	...	**8.201296e-05**	2.732854e-08	2.732854e-08	2.732854e-08	2.732854e-08	...
NN	...	7.521128e-09	7.521128e-09	7.521128e-09	7.521128e-09	**2.257091e-05**	...
NNS	...	1.670013e-08	1.670013e-08	**4.676203e-04**	1.670013e-08	1.670013e-08	...
VB	...	3.779036e-08	3.779036e-08	3.779036e-08	3.779036e-08	3.779036e-08	...
RB	...	3.226454e-08	**6.456135e-05**	3.226454e-08	3.226454e-08	3.226454e-08	...
RP	...	3.723317e-07	3.723317e-07	3.723317e-07	**3.723317e-07**	3.723317e-07	...
...	...	...	...	...	...	...	...

### Exercise 04 **Instructions:** Implement the `create_emission_matrix` below that computes the `B` emission probabilities matrix. Your function takes in α, the smoothing parameter, `tag_counts`, which is a dictionary mapping each tag to its respective count, the `emission_counts` dictionary where the keys are (tag, word) and the values are the counts. Your task is to output a matrix that computes equation 4 for each cell in matrix `B`.

```
In [ ]: # UNQ_C4 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
        # GRADED FUNCTION: create_emission_matrix

        def create_emission_matrix(alpha, tag_counts, emission_counts, vocab):
            '''
            Input:
                alpha: tuning parameter used in smoothing
                tag_counts: a dictionary mapping each tag to its respective count
                emission_counts: a dictionary where the keys are (tag, word) and the values ar
                vocab: a dictionary where keys are words in vocabulary and value is an index.
                       within the function it'll be treated as a list
            Output:
                B: a matrix of dimension (num_tags, len(vocab))
            '''

            # get the number of POS tag
            num_tags = len(tag_counts)
```

```
    # Get a list of all POS tags
    all_tags = sorted(tag_counts.keys())

    # Get the total number of unique words in the vocabulary
    num_words = len(vocab)

    # Initialize the emission matrix B with places for
    # tags in the rows and words in the columns
    B = np.zeros((num_tags, num_words))

    # Get a set of all (POS, word) tuples
    # from the keys of the emission_counts dictionary
    emis_keys = set(list(emission_counts.keys()))

    ### START CODE HERE (Replace instances of 'None' with your code) ###

    # Go through each row (POS tags)
    for i in range(num_tags): # complete this line

        # Go through each column (words)
        for j in range(num_words): # complete this line

            # Initialize the emission count for the (POS tag, word) to zero
            count = 0

            # Define the (POS tag, word) tuple for this row and column
            key =  (all_tags[i],vocab[j])

            # check if the (POS tag, word) tuple exists as a key in emission counts
            if key in emission_counts.keys(): # complete this line

                # Get the count of (POS tag, word) from the emission_counts d

                count = emission_counts[key]

            # Get the count of the POS tag
            count_tag = tag_counts[all_tags[i]]

            # Apply smoothing and store the smoothed value
            # into the emission matrix B for this row and column
            B[i,j] = (count + alpha) / (count_tag+ alpha*num_words)

    ### END CODE HERE ###
    return B
```

```
In [ ]: # creating your emission probability matrix. this takes a few minutes to run.
        B = create_emission_matrix(alpha, tag_counts, emission_counts, list(vocab))
```

```
print(f"View Matrix position at row 0, column 0: {B[0,0]:.9f}")
print(f"View Matrix position at row 3, column 1: {B[3,1]:.9f}")

# Try viewing emissions for a few words in a sample dataframe
cidx  = ['725','adroitly','engineers', 'promoted', 'synergy']

# Get the integer ID for each word
cols = [vocab[a] for a in cidx]

# Choose POS tags to show in a sample dataframe
rvals =['CD','NN','NNS', 'VB','RB','RP']

# For each POS tag, get the row number from the 'states' list
rows = [states.index(a) for a in rvals]

# Get the emissions for the sample of words, and the sample of POS tags
B_sub = pd.DataFrame(B[np.ix_(rows,cols)], index=rvals, columns = cidx )
print(B_sub)
```

Expected Output

```
View Matrix position at row 0, column 0: 0.000006032
View Matrix position at row 3, column 1: 0.000000720
              725      adroitly     engineers      promoted       synergy
CD   8.201296e-05  2.732854e-08  2.732854e-08  2.732854e-08  2.732854e-08
NN   7.521128e-09  7.521128e-09  7.521128e-09  7.521128e-09  2.257091e-05
NNS  1.670013e-08  1.670013e-08  4.676203e-04  1.670013e-08  1.670013e-08
VB   3.779036e-08  3.779036e-08  3.779036e-08  3.779036e-08  3.779036e-08
RB   3.226454e-08  6.456135e-05  3.226454e-08  3.226454e-08  3.226454e-08
RP   3.723317e-07  3.723317e-07  3.723317e-07  3.723317e-07  3.723317e-07
```

# Part 3: Viterbi Algorithm and Dynamic Programming
In this part of the assignment you will implement the Viterbi algorithm which makes use of dynamic programming. Specifically, you will use your two matrices, `A` and `B` to compute the Viterbi algorithm. We have decomposed this process into three main steps for you.

- **Initialization** - In this part you initialize the `best_paths` and `best_probabilities` matrices that you will be populating in `feed_forward`.
- **Feed forward** - At each step, you calculate the probability of each path happening and the best paths up to that point.
- **Feed backward**: This allows you to find the best path with the highest probabilities.

## Part 3.1: Initialization
You will start by initializing two matrices of the same dimension.

- best_probs: Each cell contains the probability of going from one POS tag to a word in the corpus.

- best_paths: A matrix that helps you trace through the best possible path in the corpus.