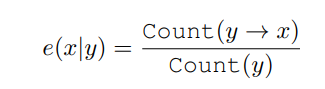
Part 1

**Estimating Emission Parameters**

Following the given formula,  


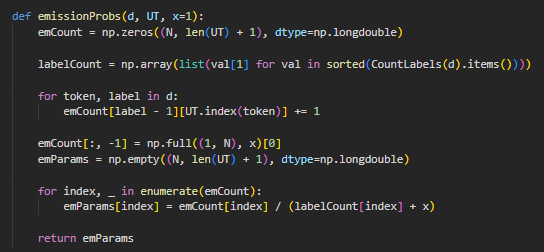
The approach that we had to adopt would be to first obtain values for number of occurrences of each label in the input data and a list of distinct unique tokens. This would be used to calculate the emission probability.



The first step is to obtain a list of unique tokens from the training data. This list is obtained using the **getUniqueTokens** function.



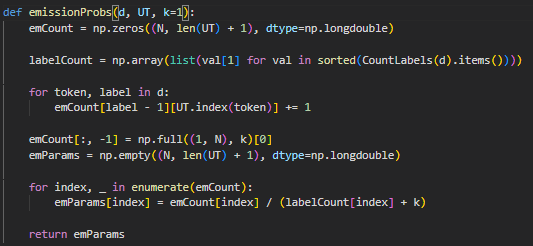
The **CountLabels** function calculates the frequency of each label in the training data. This is done using the **Counter** class, which tallies the occurrences of each label code in the training data.



The **emissionProbs** function calculates emission probabilities based on the training data and unique tokens.

1. Initialize an array **emCount** of shape (N, len(UT) + 1) to store the counts of emission occurrences, where N is the number of labels and len(UT) + 1 accounts for the additional column for the **"END"** label.
2. Calculate the total counts of each label using the **CountLabels** function.
3. Loop through each token-label pair (token, label) in the training data. This is done by indexing the list of unique tokens, UT, using the UT.index(token) function.
4. “x” is used to fill the last column of emission count, **emCount,** to account for the END label’s emission probability.
5. Initialize an array **emParams** of the same shape as **emCount** to store the calculated emission probabilities.
6. Using the given formula above, **emission probability = count of emission / (total label count + x),** we calculate the emission probabilities and store them in the **emParams** array.

**Handling of Missing Tokens from Training Set**



As seen from the code above, if the token is not found in the list of unique tokens, UT, it implies that the token is missing from the training set. Hence, k is set to 1 as a smoothing method, and added to the count for END label to handle the unseen tokens at the end of the sequence.

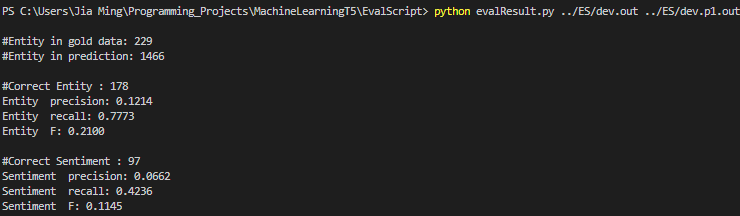
**Prediction Functions**

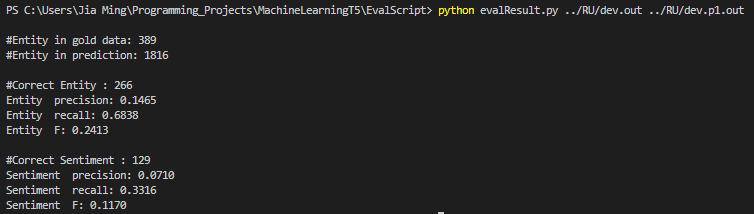
The **get\_TokenLabel(inp, UT, emParams)** function predicts the label for the token using the emission probabilities, given an input token (**inp**), a list of unique tokens (**UT**), and emission probabilities (**emParams**).

**write\_file\_predictions(lg)** function generates predictions for test data of a specific language (lg). It reads training data, calculates emission probabilities, processes test data, predicts labels using the emission probabilities, and writes the results to an output file which will then be compared against dev.out.

**Evaluation Script**

Upon running the evaluation script, the results of Part 1 ES and RU languages are as follows:



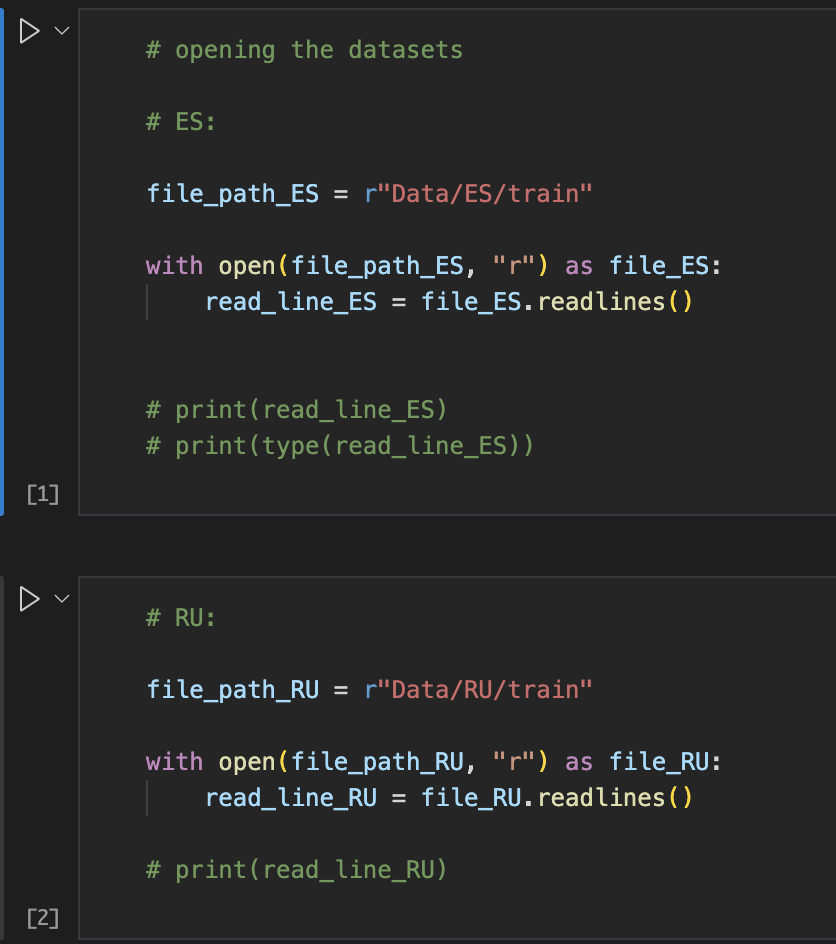


Part 2

1. **Estimate Transition Parameters:**

Opening the datasets:

Read the training data as a list with each element as a line of the training set:



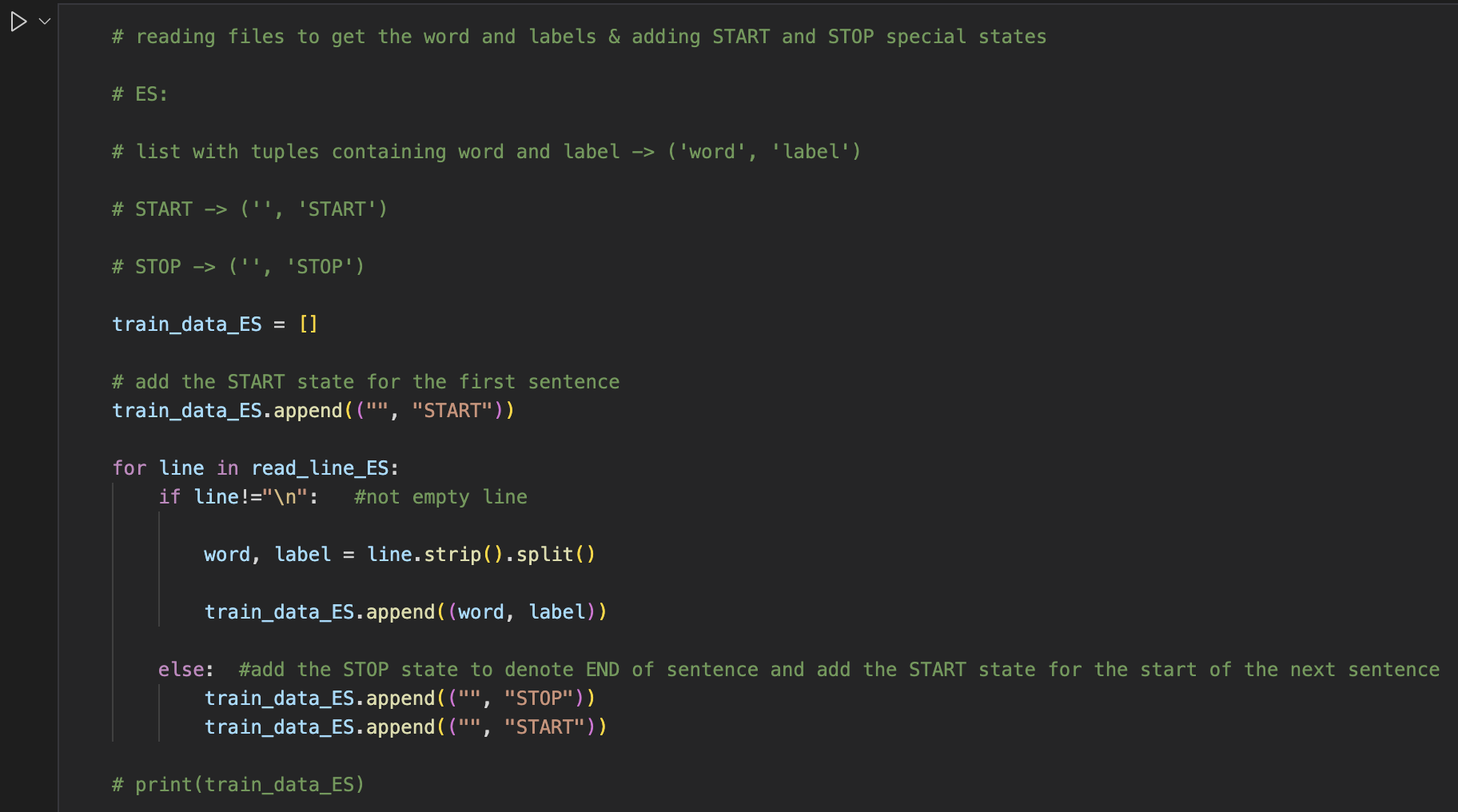
Add START and STOP special states:

START -> (‘ ‘, ‘START’)

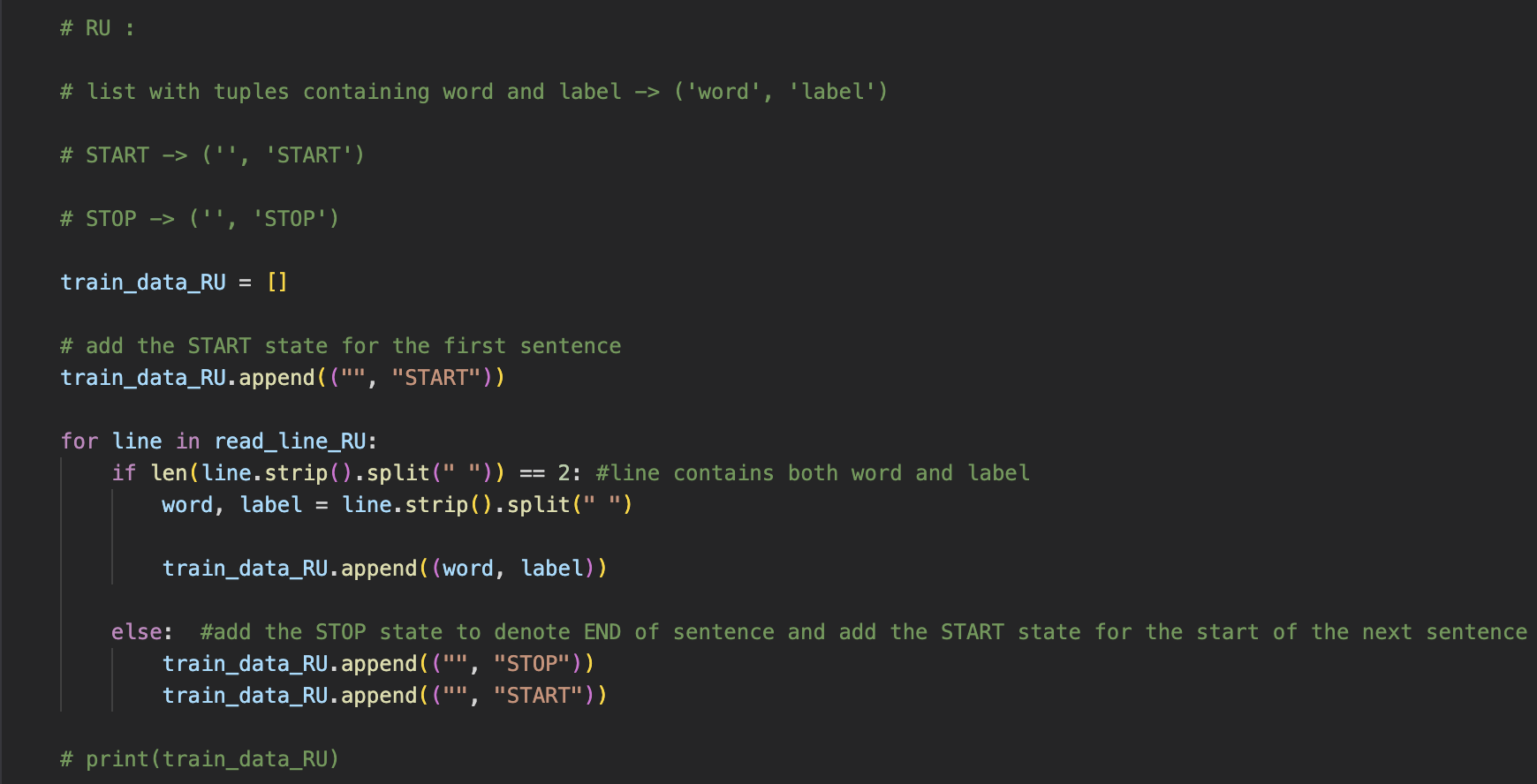
STOP -> (‘ ‘, ‘STOP’)

And split the data into word and label and append it to a list of training data examples:

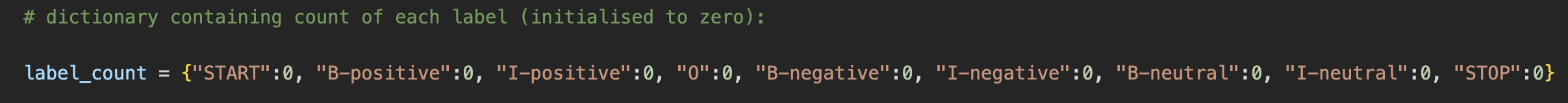
For ES:



For RU:

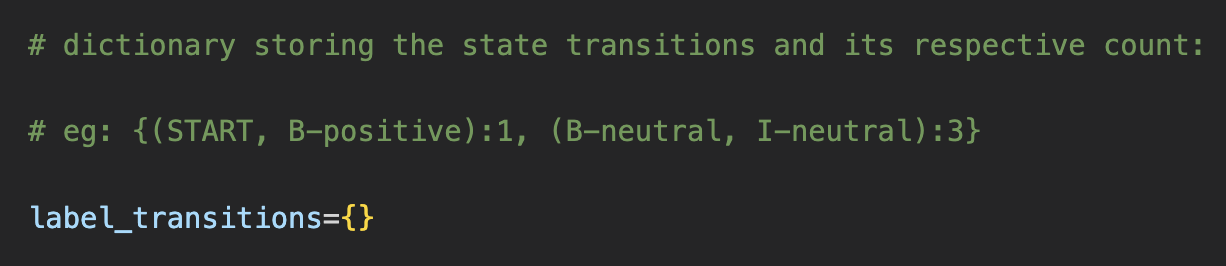


Label\_count dictionary that contains the count of each state has its value (initialised to 0) and the state as its key:



Label\_transitions dictionary that contains the state transitions as the key and its respective count as its value:

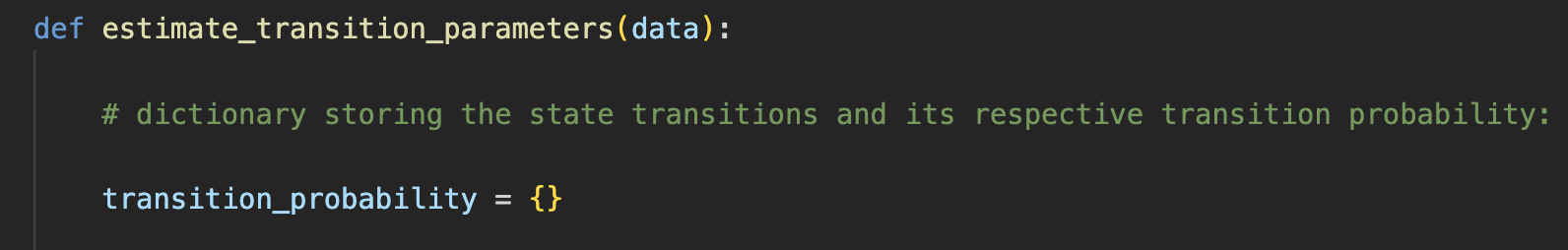
State transitions -> (prev\_state , current\_state)



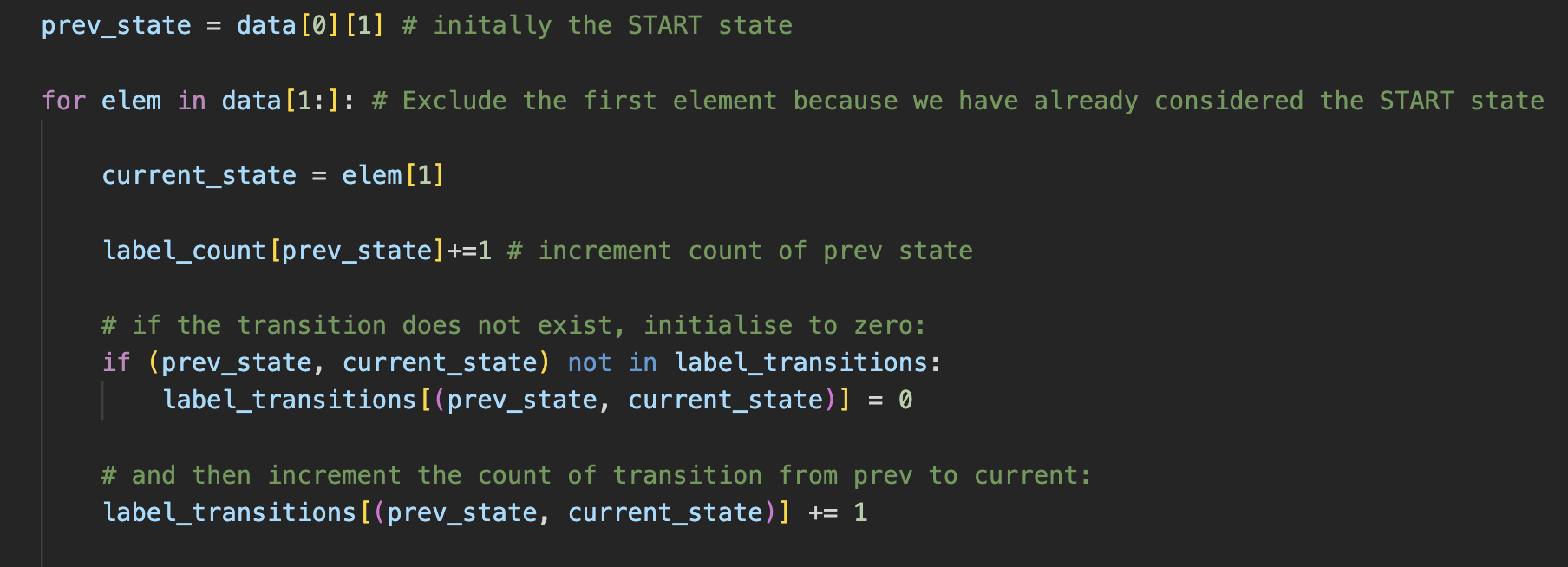
The estimate\_transition\_parameters function that takes in the dataset as its argument:

Transition\_probability dictionary is initialized to store the state transitions as the key and its respective transition probability as the value:

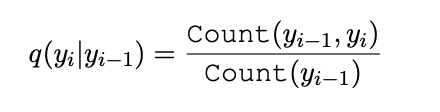
**State transition is defined as -> (current\_state, prev\_state)** according to the convention followed in the question



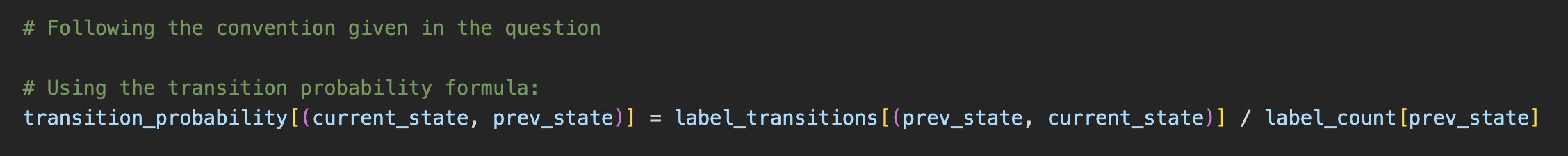
The function iterates over the data set and everytime it encounters a transition from prev\_state to current\_state, the count of the state transition is incremented by 1. Similarly, the count of the state also increases



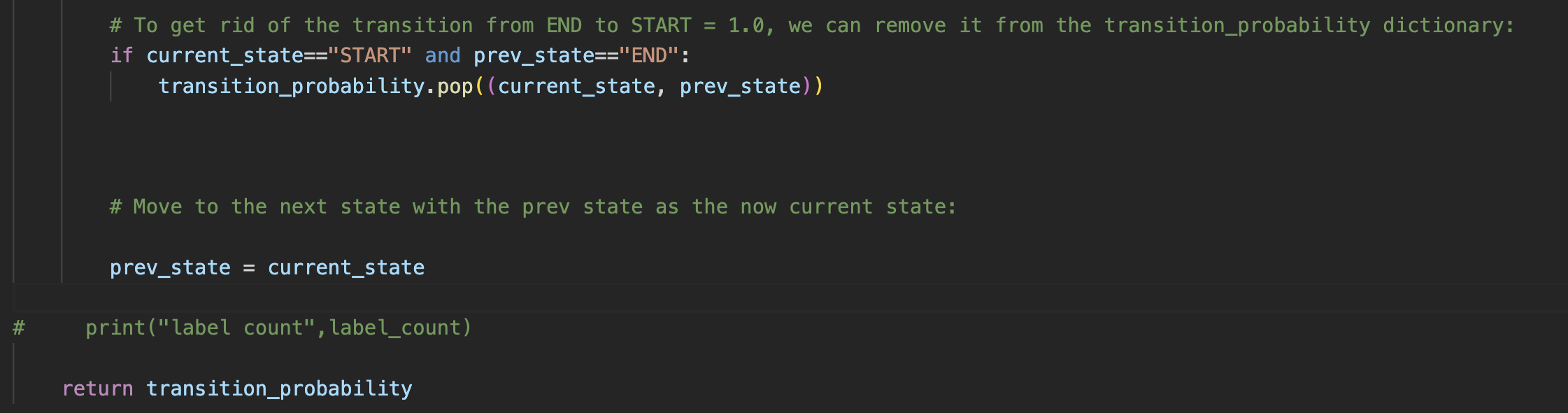
Calculate the transition\_probability using the formula:



For each state in the set of states, we calculate the transition probability by dividing the count of transition from prev\_state to current\_state by the count of prev\_state



We further cleanup the function by removing the transition from END to START after every line in the dataset:



**The results for the transition parameters for the datasets ES and RU are as follows:**

The format of the transition parameters is such that:

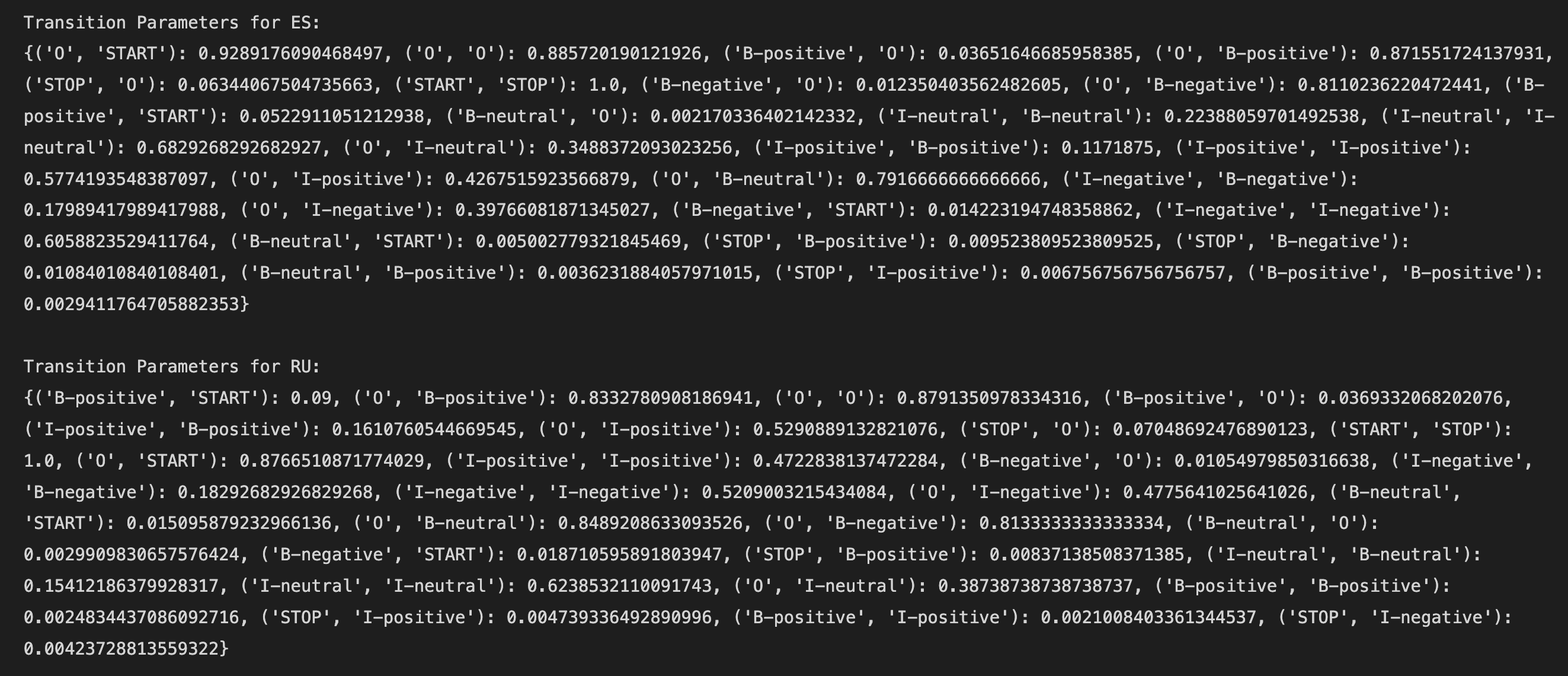
For eg, if we look at the transition parameters for the ES dataset,

transition\_probability = {('O', 'START'): 0.9289176090468497 … }

where the transition probability of state START **to** state O is 0.929

OR,

q (O| START) = 0.929



(b)

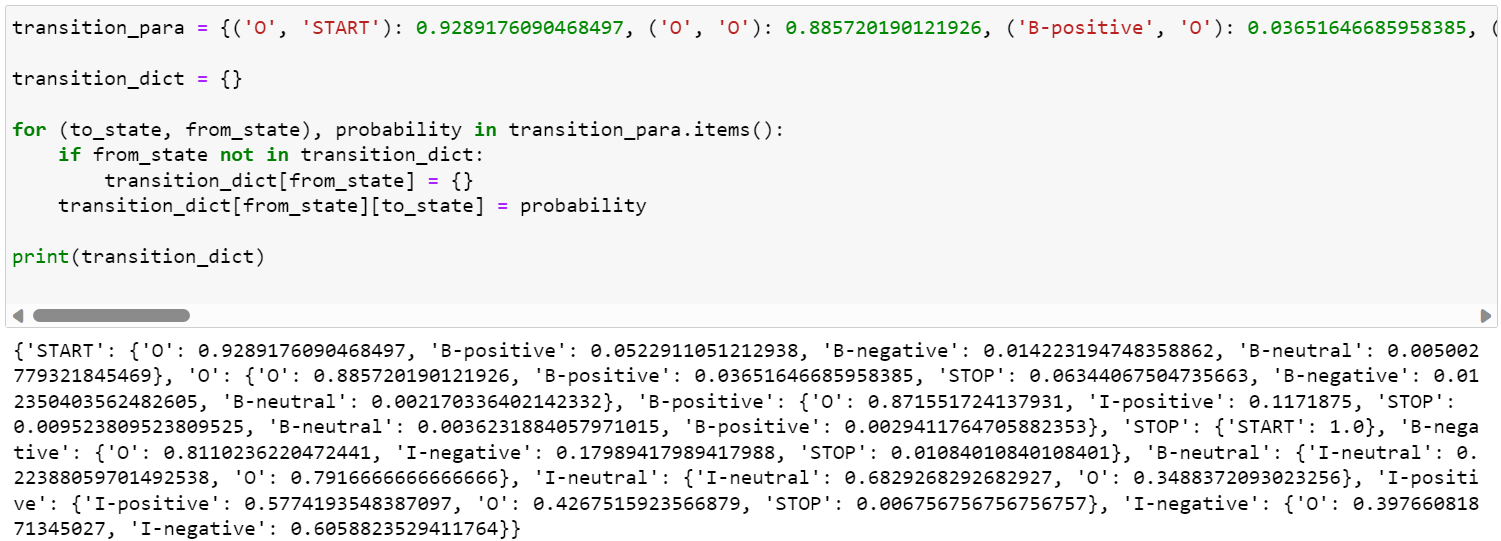
Import necessary packages and open ES/RU files respectively.



Convert calculated emission parameters to a dictionary with keys as tuples of observation and state, and values as probabilities.

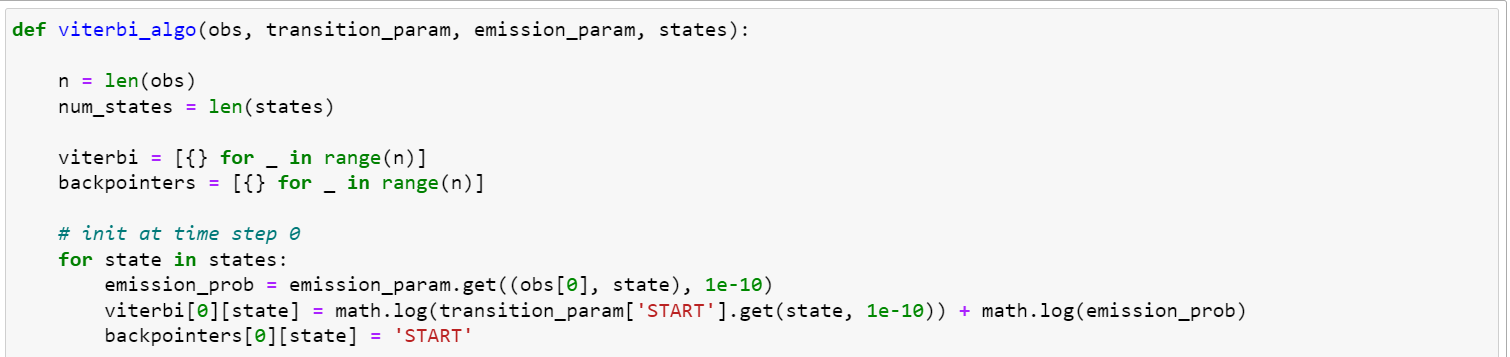


Convert transition probabilities calculated in the previous part into a dictionary format.



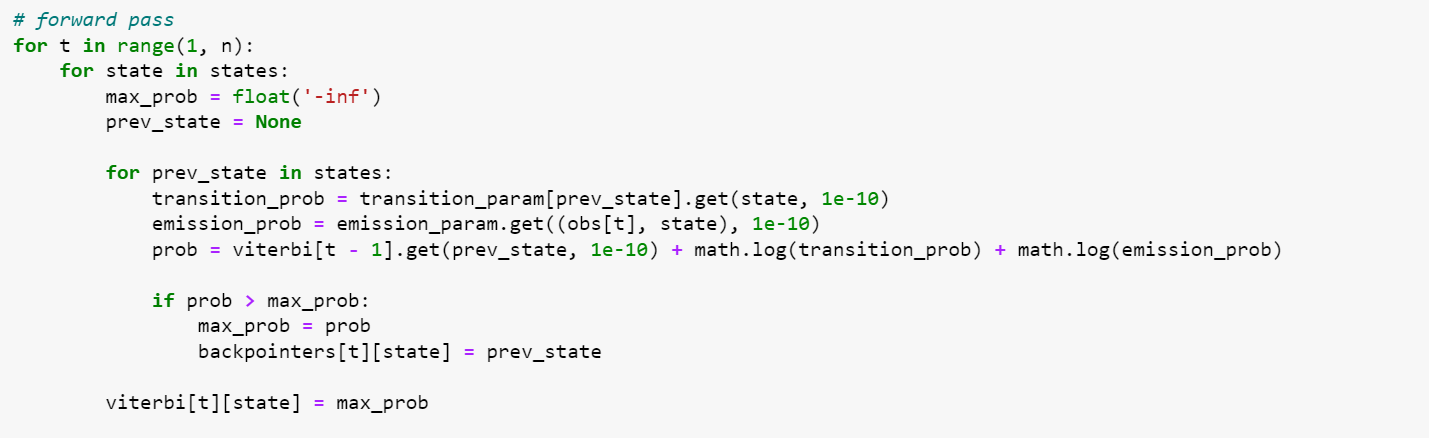
Starting off the Viterbi algorithm, after initializing, for each state in the states list, we calculate the initial score and note the backpointer.

The initial score for each state is calculated as the logarithm of the product of the initial transition probability from the 'START' state to the current state and the emission probability of the first observation given the state.



For forward pass, we iterate through time steps from 1 to n – 1 and calculate the maximum score achievable at that state and set the backpointer leading to that maximum score.

The algorithm iterates over all possible previous states and calculates the score for transitioning to the current state from each previous state. The score is the sum of the logarithms of the transition probability, emission probability of the current observation, and the score of the previous state at time step t - 1.



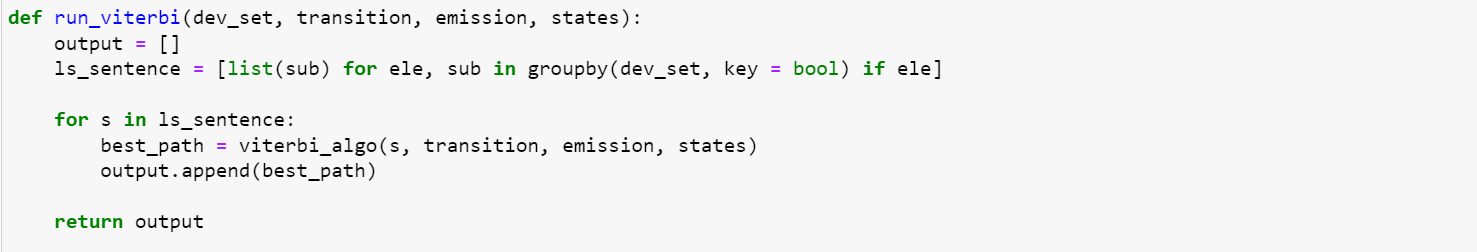
For termination step, we identify the most likely final state by calculating the maximum score across all possible states, considering the transition probability from each state to the 'STOP' state.



For the backtracking step, we start from the most likely final state obtained in the termination step, and backtrack to find the most likely sequence of hidden states. It follows the backpointers from time step n - 1 to time step 0. At each time step, the backpointer indicates the previous state that led to the maximum score.



Next, we run this Viterbi algorithm on a set of sequences (dev\_set), applying the algorithm to each sequence, and collect the resulting most likely state sequences in the output list.



Finally, we run run\_viterbi to execute the algorithm and write output to dev.p2.out respectively.



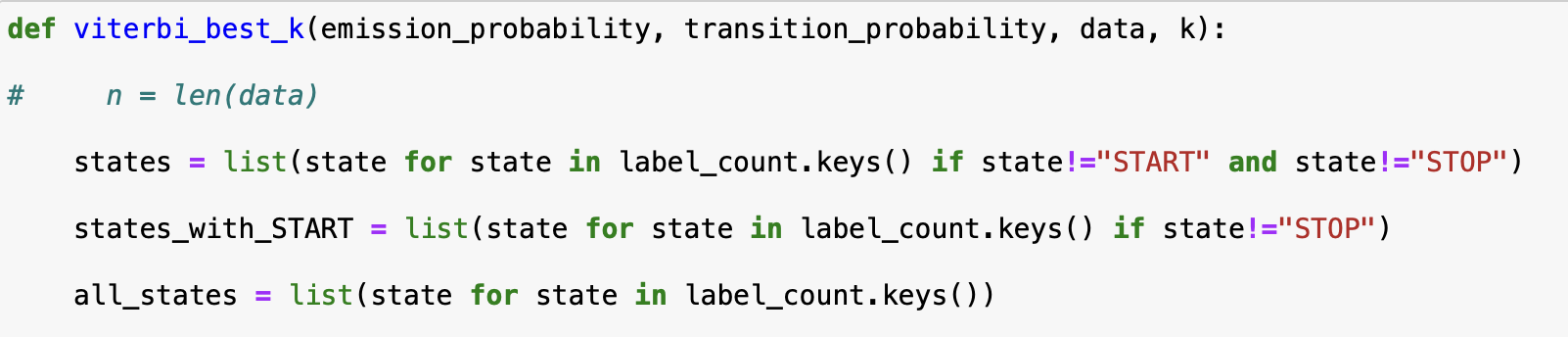
Part 3

Find the k-th best output sequences given the emission and transition parameters:

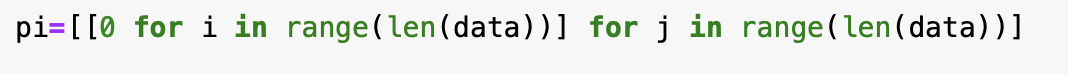
Here, states is a list of the 7 states excluding START and STOP

States\_with\_START includes the 7 states plus the START state

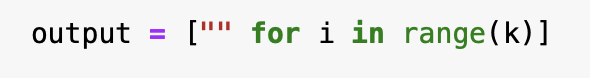
All\_states includes all 7 states plus START and STOP states:



2D array pi to store the scores at each time stamp:



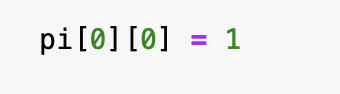
Output array to store the output sequences:



Following the Viterbi algorithm as given in the slides:

Here, START is index 0 and STOP is index n where n = 9:

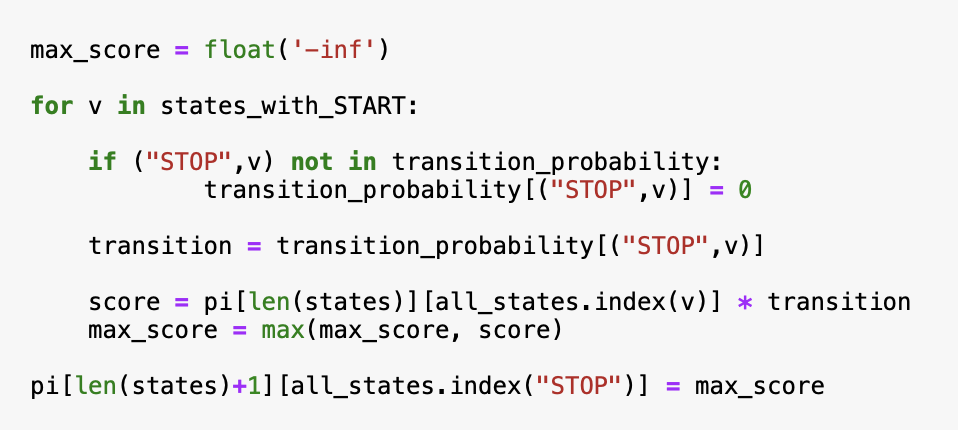
1. Initialisation Step :



1. The for loop:

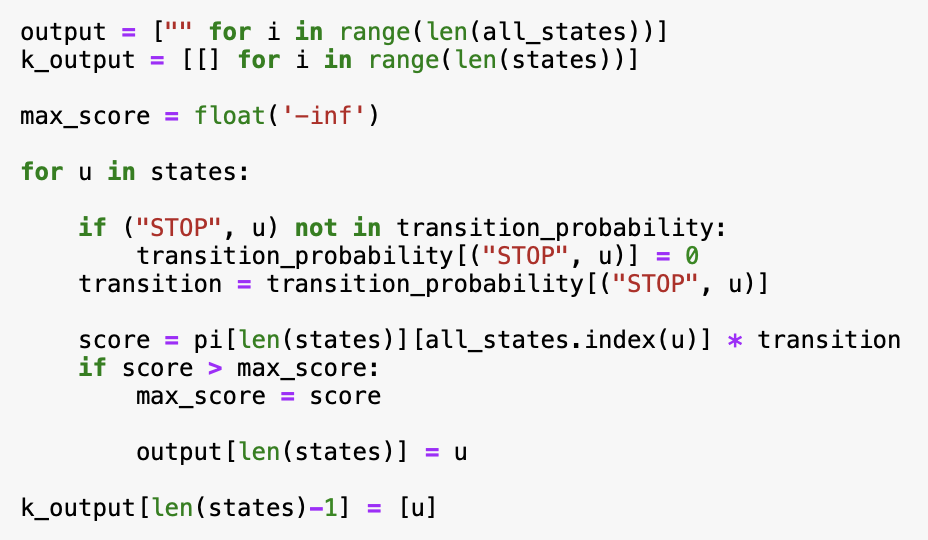


1. Final Step for the STOP state:

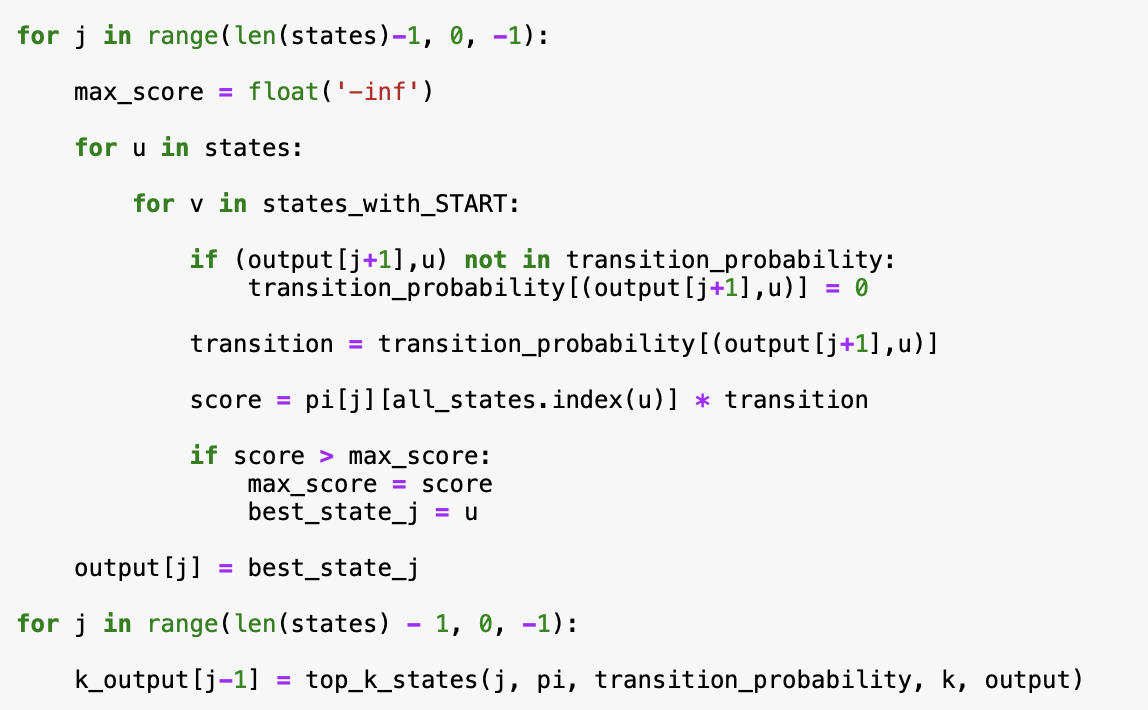


Backtracking step to get the path from the scores:

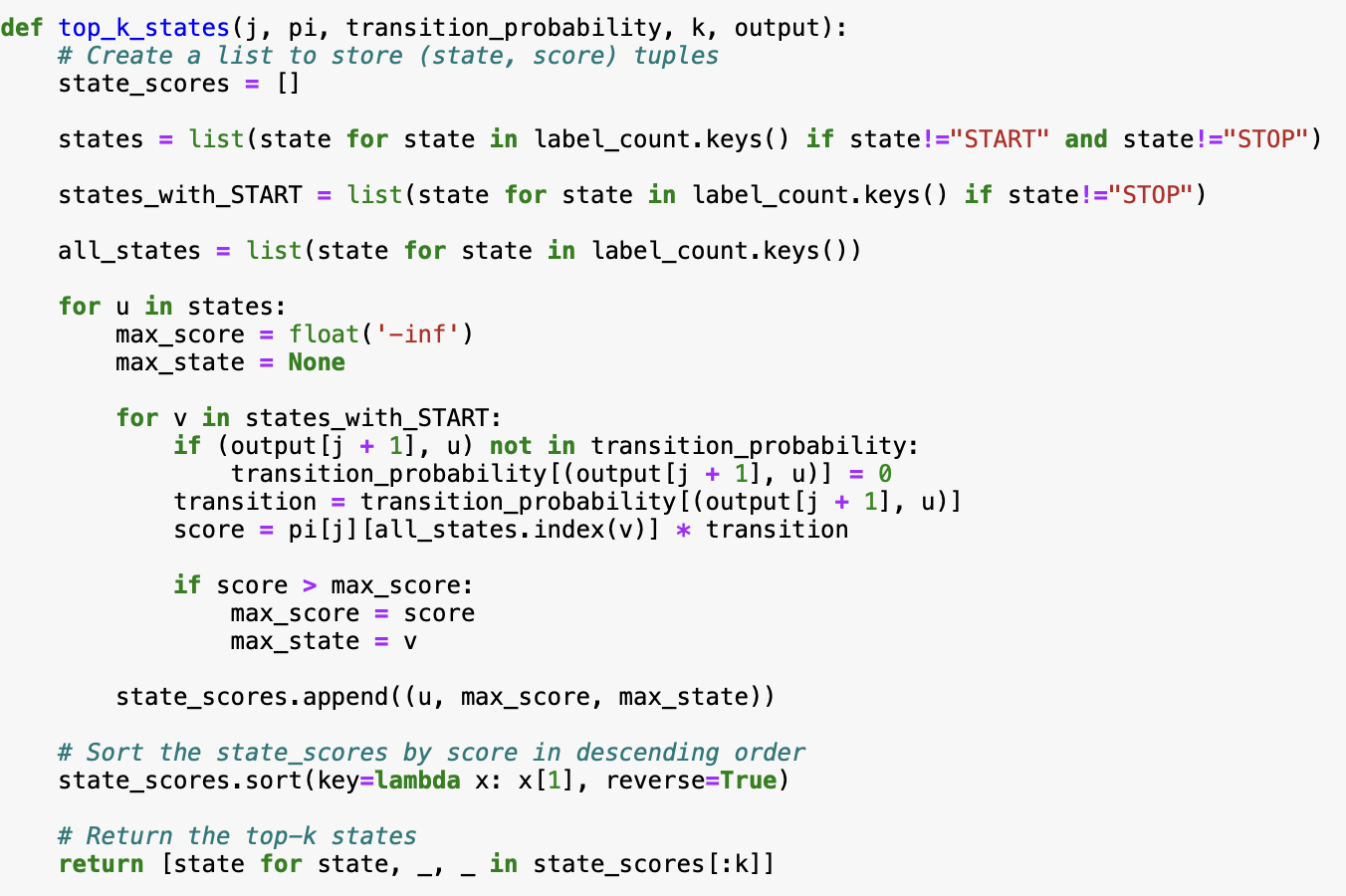
1. Calculate the optimal state for Yn:



1. Calculate the optimal state values for Yi upto n-1:



Algorithm to find the k-th best output sequences by backtracking through each observation and selecting the best optimal path of states:



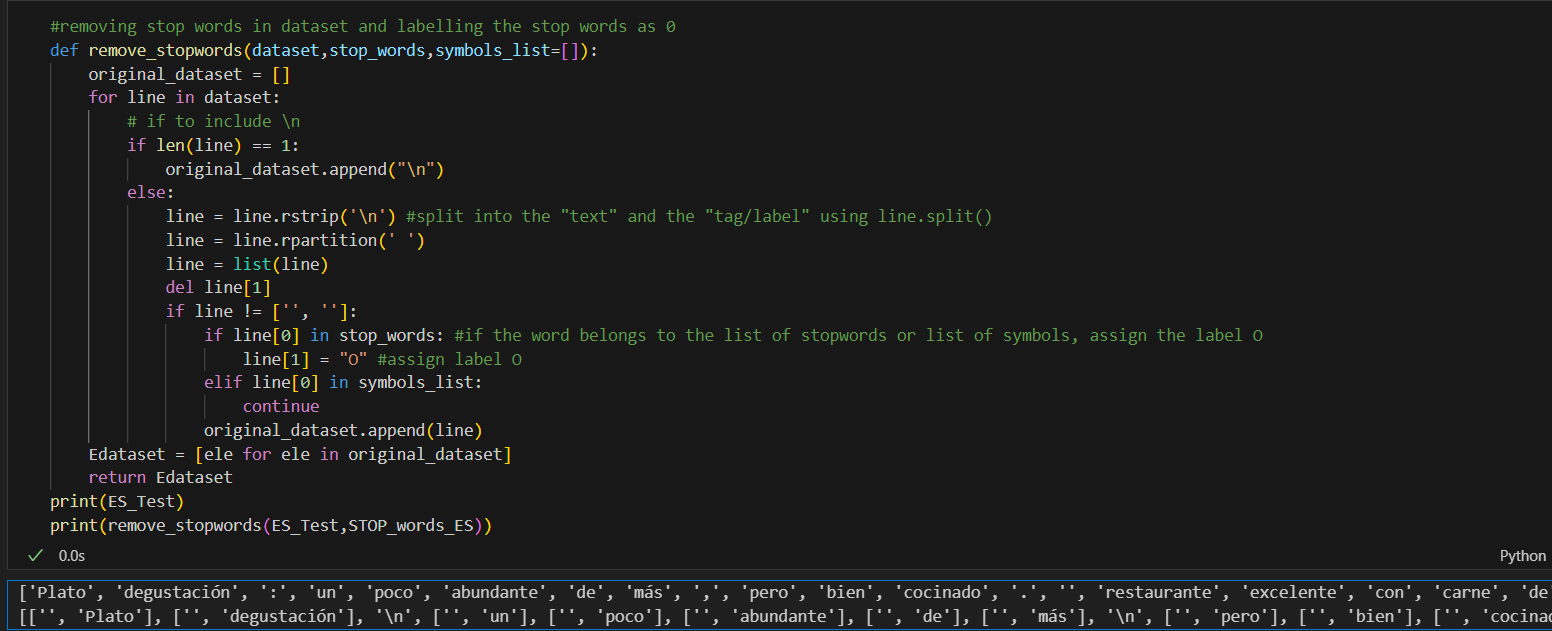
Part 4

Based on the training and development set, it is necessary to improve the accuracy of the Viterbi Algorithm.

Approach taken:

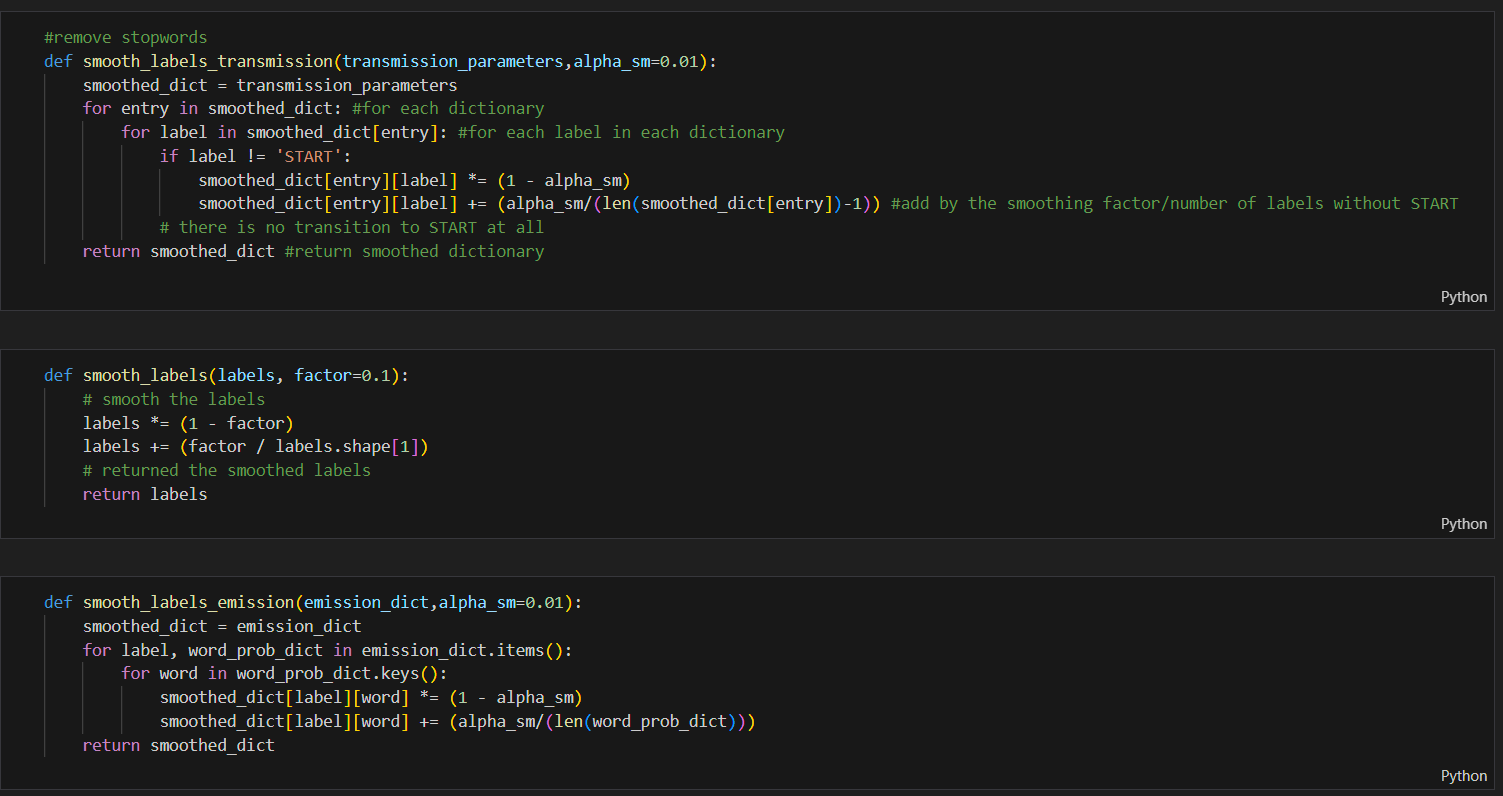
1. Data Processing

Before going through the dataset, we read and remove the stop words for the RU and ES datasets. Stop words are a list of frequently used words that can be divided into language, time, location and number-based stop words. The removal of stop words does not change the meaning of the sentences in the dataset but rather improves the time complexity as we run the model on longer word sequences.

* + We utilize an open-source file that can be found on GitHub which contains the stop words in the respective languages.
    - ES stop words: <https://github.com/stopwords-iso/stopwords-es/blob/master/stopwords-es.txt>
      * + Ru stop words: <https://github.com/stopwords-iso/stopwords-ru/blob/master/stopwords-ru.txt>
  + Function *read\_stopword*s is used to convert stop words found from the open-source file into a list that will be used in future functions and smoothing process.
  + Function *remove\_stopwords* is used to remove the stop words from the data set and label these stop words to ‘O’.
  + Function *get\_symbol* is used to check non alphanumeric characters such as symbols from the dataset and appending it into a list.

1. Label Smoothing

Label smoothing is used for emission and transition parameters of the Hidden Markov Model and it is used to prevent overfitting of the model itself. This is have the most probable label to have highest probability and have others have a very small probability.



* + Function smooth\_label\_transition is used to remove stopwords
  + Fun