50.007 Machine Learning Sentiment Analysis Project

Riley Chin 1004147

Tan Hong Yew 1004235

Yik Khuen 1004148

[**Running the codes**](#_9k8vtr5cpspk) **3**

[**Note**](#_pmkf08sv4vcx) **3**

[**1. Question 1**](#_tuyxs2i6mcdt) **4**

[1.1 Preprocessing](#_h1ozimndm4ib) 4

[1.1.1 get\_all\_text\_tags()](#_ug95rslo774w) 4

[1.1.2 create\_words\_tag\_dict()](#_c8g7npe5lo5y) 5

[1.2 Emission MLE and special case MLE](#_reeviawszmfk) 6

[1.3 Simple sentiment analysis system](#_6afj8xiq56jc) 7

[1.4 Results](#_27upkok3c8qm) 8

[**2. Question 2**](#_1222wq7myfct) **9**

[2.1 Preprocessing](#_t50g5iyrg67g) 9

[2.2 Transition MLE](#_a64193mhfq6x) 11

[2.3 Viterbi](#_46vzvkwo2uyq) 11

[2.4 Results](#_ey7q5dam84p2) 14

[**3. Question 3**](#_oarj2sdz99ij) **16**

[3.1 Preprocessing](#_9ouk0m7lv9f) 16

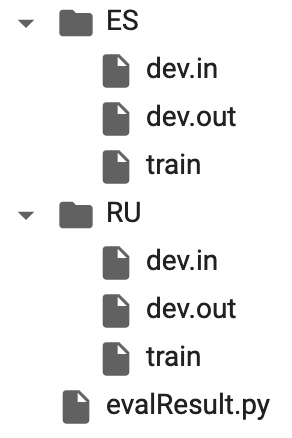
[3.2 Viterbi 5th best algorithm](#_dn2189ltk08i) 16

[3.3 Results](#_qeoowweuzfir) 19

# Running the codes

* [Github Link](https://github.com/rileychin/50.007-ML-project)
* Create folders for ES and RU
* Put train, dev.in, dev.out into their respective folders
* Put evalresult.py into the same directory as the ES and RU folders

Directory should look like this:



For each part of this project, run every cell sequentially. No extra input is necessary.

# Note

For this report, we will encounter the term “tag” and “state” often. Tag represents the possible states, namely:

* O
* B-neutral
* B-negative
* B-positive
* I-neutral
* I-negative
* I-positive
* START (when needed)
* STOP (when needed)

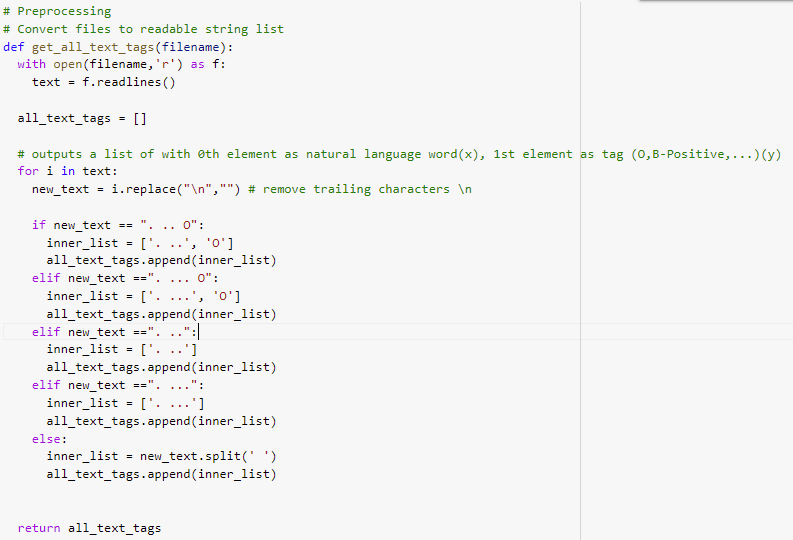
These terms are both used interchangeably.

# 1. Question 1

## 1.1 Preprocessing

The preprocessing is done in several stages. From the raw input text file, we have several desired outputs like the total tag counts, and the words associated with tags dictionary.

### 1.1.1 get\_all\_text\_tags()



*Fig 1.1.1.1: get\_all\_text\_tags() code block*

* Description:
  + Converts the wall of text from a file to the corresponding 0th index = word, 1st index = tag. Special cases for the RU dataset are identified via hardcode.
* Input:
  + Filename
* Output:
  + A list of lists with 0th index = word, 1st index = tag.

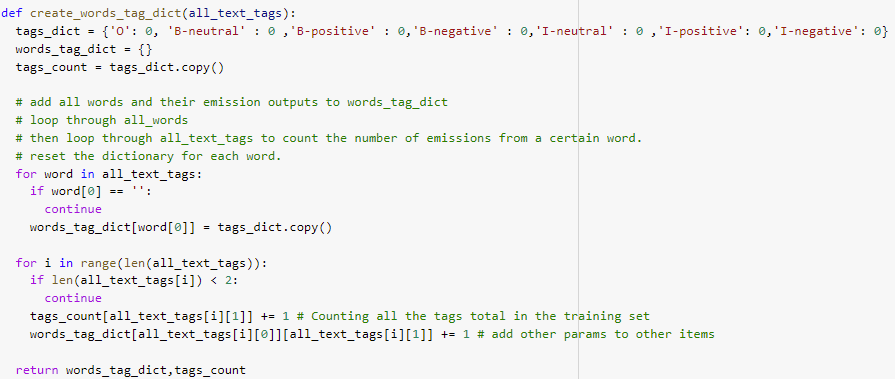
The output of this function is as such:



*Fig 1.1.1.2: get\_all\_text\_tags() output*

This function is used for all subsequent preprocessing parts and is available in the different notebooks for the other questions as well.

### 1.1.2 create\_words\_tag\_dict()



*Fig 1.1.2.1: create\_words\_tag\_dict() code block*

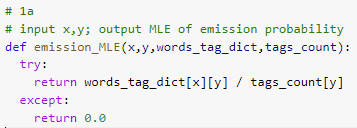
* Description:
  + Converts the list of lists to 2 associated dictionaries. The total count for the tags, and the words associated with the tags.
* Input:
  + List of lists generated by *get\_all\_text\_tags()*
* Output:
  + A dictionary with the total count for the tags, and the associated words for the tags.



*Fig 1.1.2.2: output of create\_words\_tag\_dict()*

## 

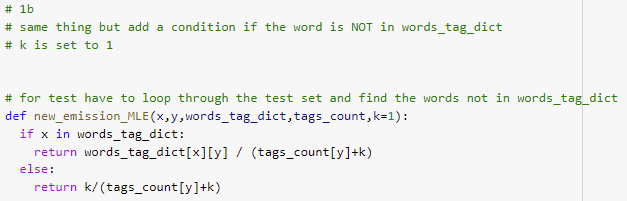
## 1.2 Emission MLE and special case MLE



*Fig 1.2.1: emission\_MLE() code block*

* Description:
  + Calculates the emission probability based on the formula
* Input:
  + Word *x*, tag *y*, words\_tag\_dictionary, tags\_count dictionary
* Output:
  + Float value representing emission probability of word *x* from tag *y*

This is a simple implementation of the formula for question 1a. The ‘try’ surrounds the base case while the ‘except’ returns 0.0, when the inputs are invalid or when tags\_count returns 0.



*Fig 1.2.2: new\_emission\_MLE() code block*

* Description:
  + Calculates the new emission probability based on the formula for unknown words.
* Input:
  + Word *x*, tag *y*, words\_tag\_dictionary, tags\_count dictionary
* Output:
  + Float value representing the emission probability of word *x* from tag *y*

This is another simple implementation of the formula if word token x appears in the training set. if word token x is the special token #UNK#

## 1.3 Simple sentiment analysis system



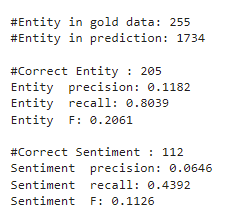
*Fig 1.3.1: get\_argmax() code block*

* Description:
  + Gets the associated tag for the highest probability of emission from the given word.
* Input:
  + Word *x*, words\_tag\_dictionary, tags\_count dictionary
* Output:
  + Tag with the highest probability of emitting word *x*

Created a function which loops through each tag and returns the tag with the highest MLE for a given word.

Run the function for every word to get the predicted tag, and create the predicted output file.

## 1.4 Results



*Fig 1.4.1: Results for ./ES/dev.in*

For the simple sentiment analysis approach, we obtained a total of 1734 predicted entities. Although we obtained a rather high recall, the precision and F score is rather low.

**

*Fig 1.4.2: Results for ./RU/dev.in*

Similar to the ES dataset. We have a high recall and low precision and F score. This could be because we have more false positives than false negatives in the predictions. There are alot of negative predictions that can become false positives as opposed to positives becoming false negatives.

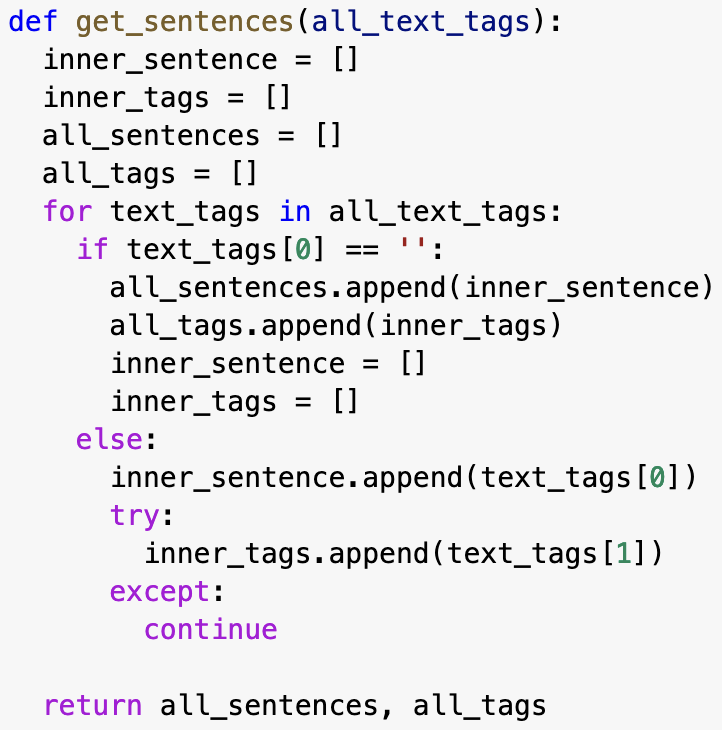
# 2. Question 2

## 2.1 Preprocessing

Similar preprocessing as part 1, but with an additional layer of lists to separate each sentence.

We also formatted the information as dictionaries containing dictionaries for easy access ie. We prepared dictionaries of transition MLEs (separated initial from the rest) and emission MLEs for each tag.

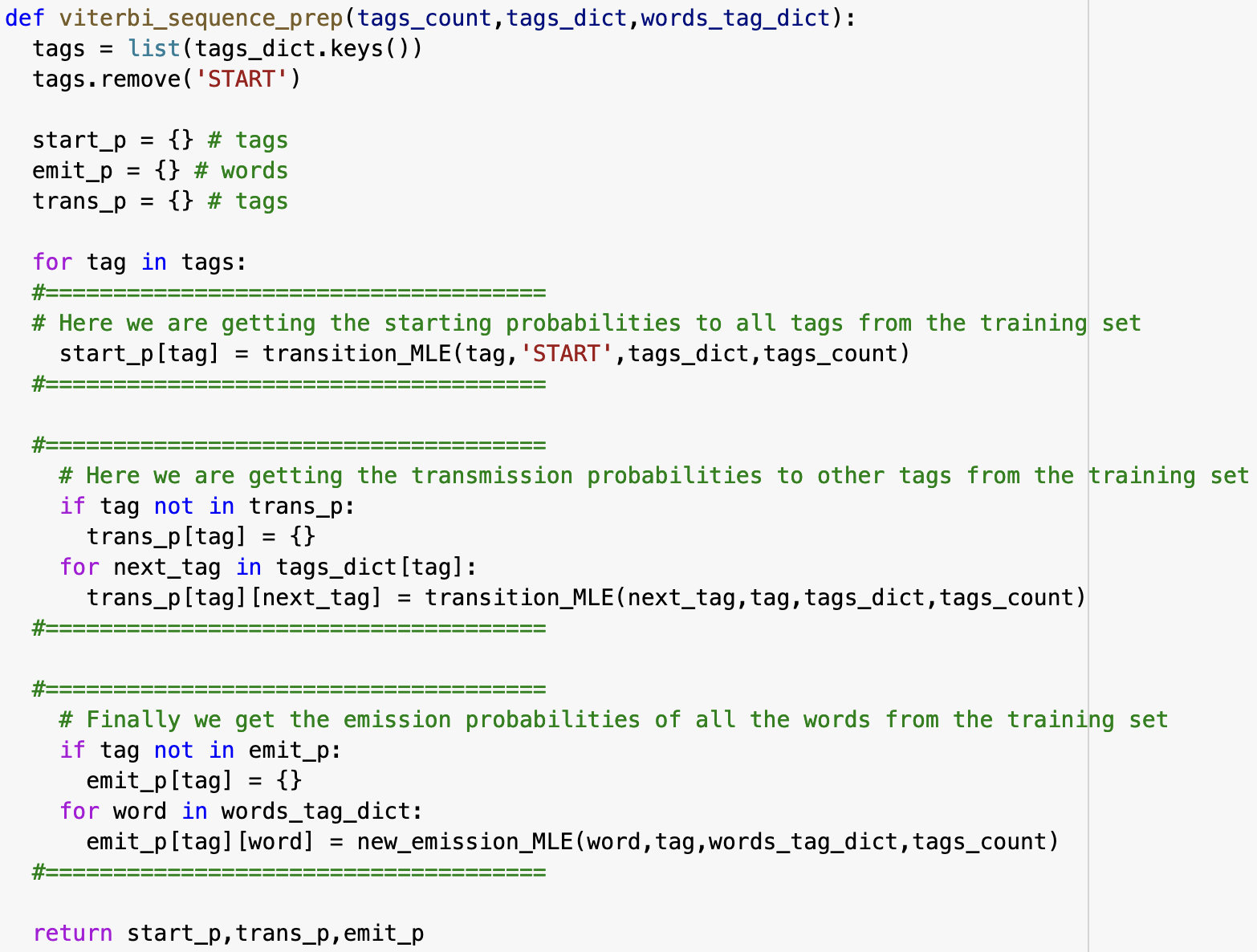
2.1.1 get\_sentences()



*Fig 2.1.1.1: get\_sentences() code block*

* Description:
  + Converts our data to a list, with inner lists for each sentence containing each word of the sentence as individual items
* Input:
  + all\_text\_tags
* Output:
  + A list with inner lists for each sentence word, and a list with inner lists for each sentence tag if applicable (labelled set).

2.1.2 viterbi\_sequence\_prep() \*this code block is just before our viterbi sequence



*Fig 2.1.1.2: viterbi\_sequence\_prep() code block*

* Description:
  + Produces dictionaries containing the initial transition probability, transition probability and emission probability of all the tags and words
* Input:
  + Tags\_count, tags\_dict, words\_tag\_dict
* Output:
  + Dictionaries for starting transition probability, other transition probability and emission probability.

## 2.2 Transition MLE

## 

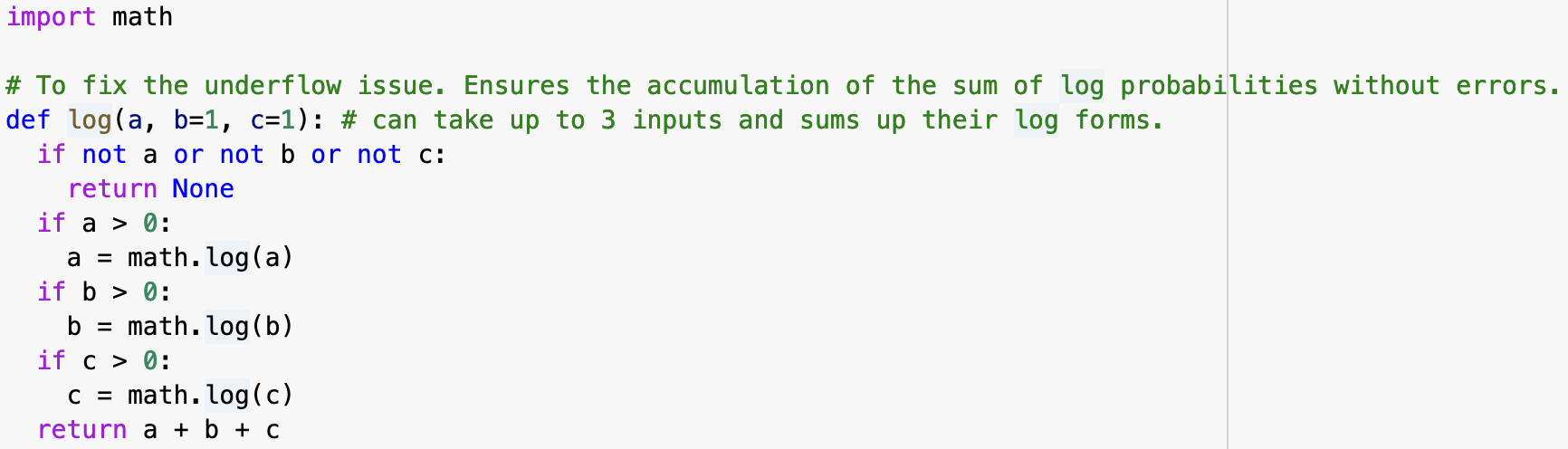
*Fig 2.2.1: transition\_MLE() code block*

* Description:
  + Calculates the transition probability based on the formula.
* Input:
  + Current tag (x), previous tag *(x-1)*, tags\_dictionary, tags\_count dictionary
* Output:
  + Float value representing the transition probability from previous\_tag to current\_tag

This is a simple implementation of the formula for question 2a. The ‘try’ surrounds the base case while the ‘except’ returns 0.0, when the inputs are invalid or when tags\_count returns 0.

## 2.3 Viterbi

2.3.1 log()



*Fig 2.3.1: log() code block*

* Description:
  + Function to obtain log-values of 1, 2 or 3 inputs and sum them up.
* Input:
  + Float value a, with optional b and c.
* Output:
  + Sum of log values of a, with b and c if included

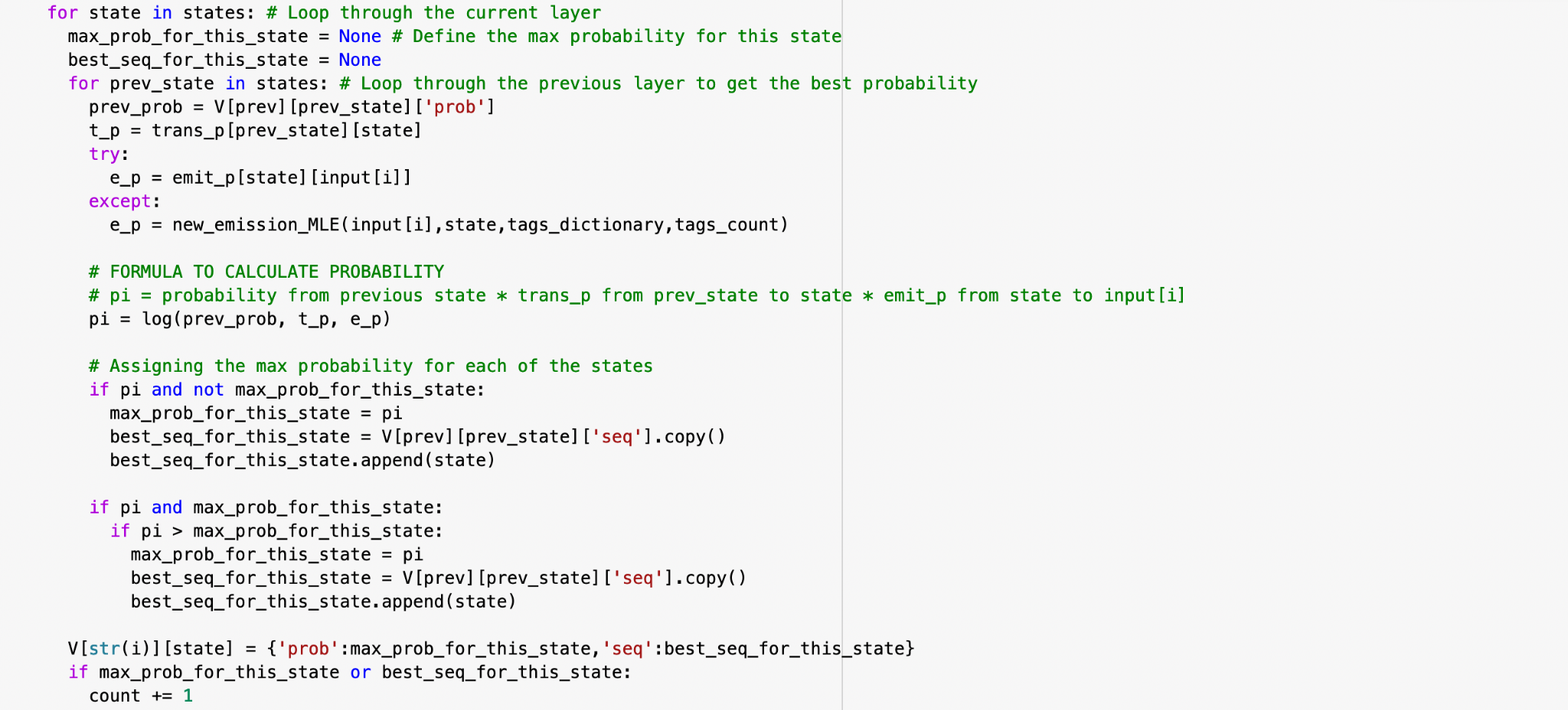
To prevent underflow, we used log-probability for every probability value. We created a function to help us compute the sum of log probabilities and the way it was coded allowed for a variable number of arguments.

2.3.2 viterbi\_2() \*named as such because we had a first iteration which did not work.

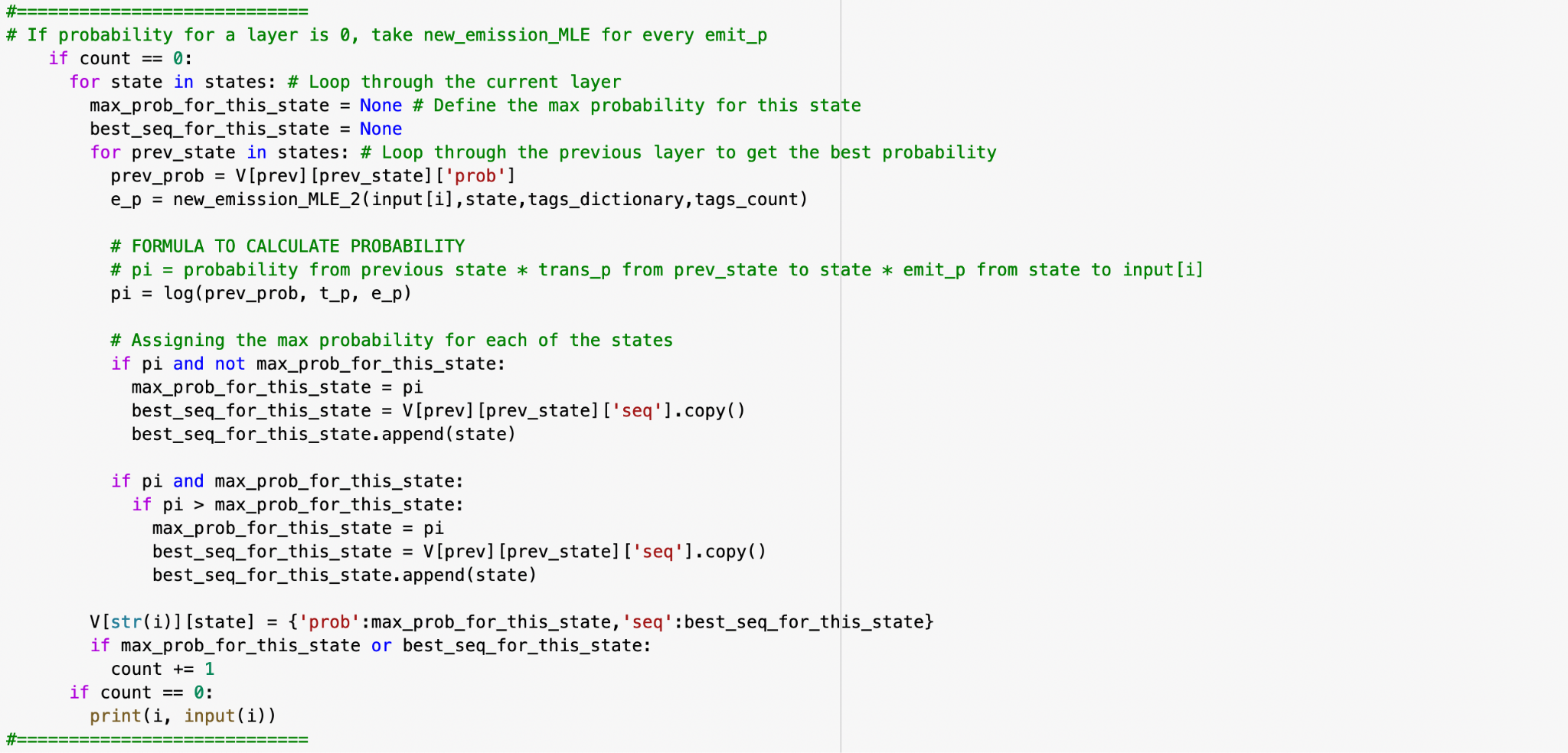
The viterbi sequence code consists of a few parts which we broke down and explained below:



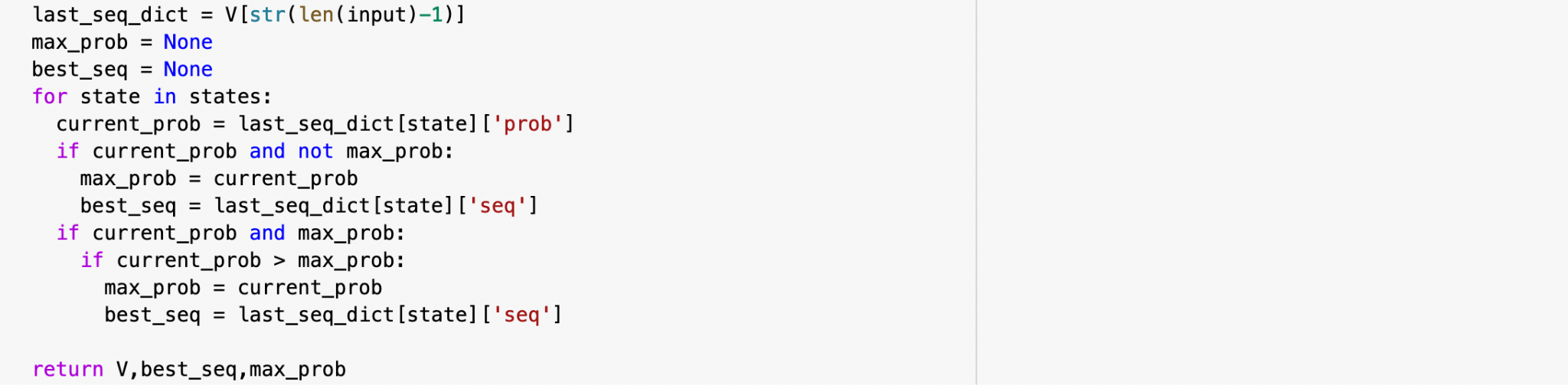
*Fig 2.3.2.1: viterbi\_2() code block initial stage*



*Fig 2.3.2.2: viterbi\_2() code block main body*



*Fig 2.3.2.3: viterbi\_2() code block special case workaround*



*Fig 2.3.2.4: viterbi\_2() code block final stage*

* Description (Initial stage):
  + Processes the transition from START state to the first state
* Description (Main body):
  + Processes the transition from state *yi* to state *yi+1*
* Description (Special case):
  + Accounts for special cases where the probability value for every node of a layer is 0, due to test data containing a sequence not found in training data
* Description (Final stage):
  + Acquire the best sequence result and probability for that sequence from the last layer of the dictionary
* Input:
  + Input list, start\_p, trans\_p, emit\_p, tags\_dictionary, tags\_count
* Output:
  + viterbi sequence dictionary, best sequence for input, max probability for that best sequence

Created a function based on the viterbi algorithm. We removed the need for a backtracking stage by storing the best sequence for every layer alongside the highest probability.

One issue we had was that in one layer of the Viterbi algorithm for some sentences, the best probability for each state is 0. With there being no best probability, the algorithm cannot find the next best sequence and produces a None output. To solve this, we tested 3 solutions for every state when a layer has 0 best probability.

* Solution 1: Use the new\_emission\_MLE formula (from part 1) for every state.
* Solution 2: Take previous\_prob \* transition\_prob for every state.
* Solution 3: Take previous\_prob \* emission\_prob for every state.

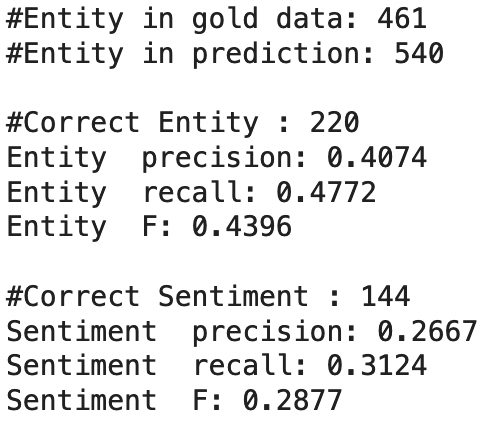
We decided to use solution 1 as pi is still calculated by previous\_probability \* transition\_prob \* emission\_prob which adheres to the algorithm and part 1 the most.

## 2.4 Results

## 

*Fig 2.4.1: Results for ./ES/dev.in*

Our precision and F-score values are higher than for question 1. This was expected as the viterbi algorithm also took the transition probability into account rather than only the emission probability like in question 1, which would make sense as the sequence of words in a sentence matters and should be considered.



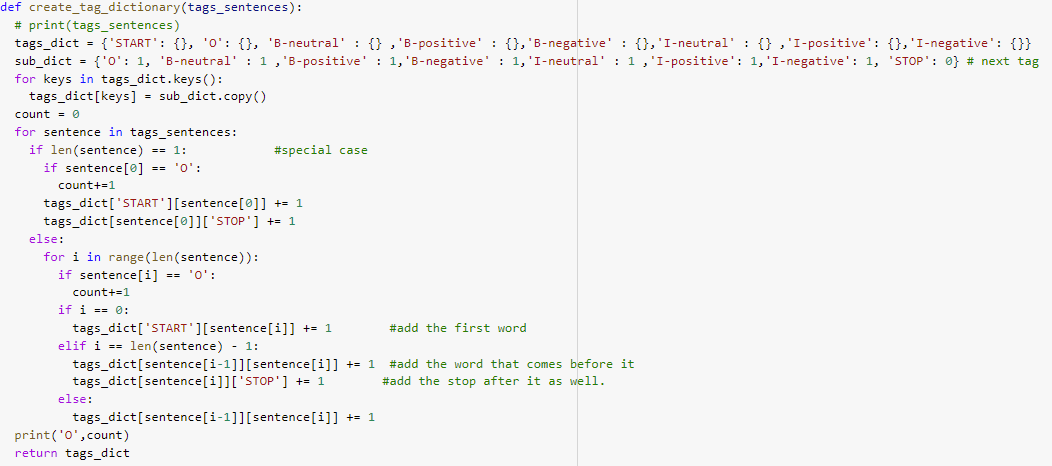
*Fig 2.4.2: Results for ./RU/dev.in*

Similar to the ES dataset, the precision and F-score values are higher than that of question 1.

# 3. Question 3

## 3.1 Preprocessing

The preprocessing for this question is the same as Question 2. The algorithm we will use takes in the same parameters. However, for this question, we are experimenting with pseudocounts. This means that all transition probabilities from state *u* to state *v* have their counts all increased by 1. This will ensure there is a possibility that there will always be a 5th best path to consider in the algorithm. The changes are made as follows:

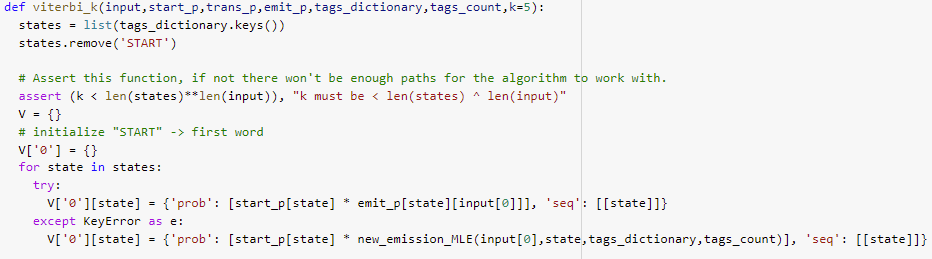


*Fig 3.1.1: Updated create\_tag\_dictionary() with pseudocounts code block*

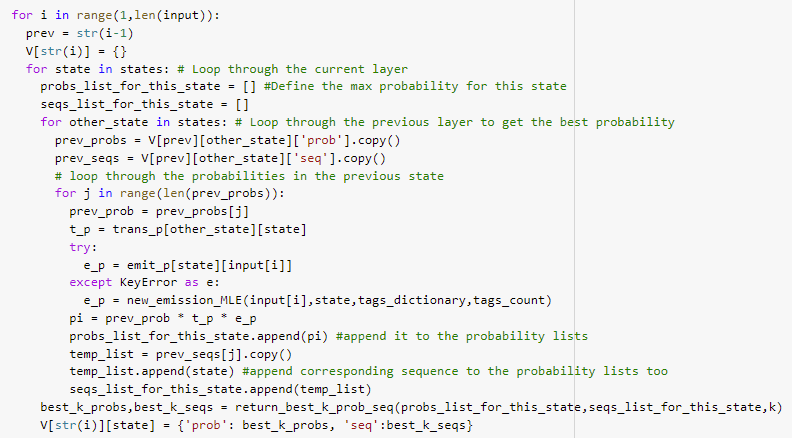
For all the tags other than STOP. We increment the transition count by one. This method also removes to account for when some paths encounter 0 probability at some layers. We do not do this for STOP as we want a consistent stopping criteria with the previous question.

## 3.2 Viterbi 5th best algorithm

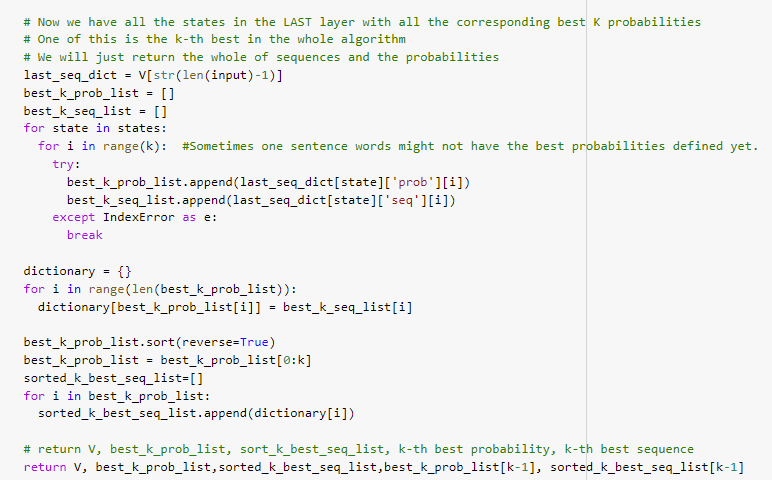
Using the same start, emission, transition probabilities, we then only had to change the dynamic programming portion of the algorithm to store all *k* best possible paths at each node for each layer. What follows is the following algorithm:



*Fig 3.2.1: viterbi\_k() code block initial stage*



*Fig 3.2.2: viterbi\_k() code block main body*



*Fig 3.2.3: viterbi\_k() code block final stage*

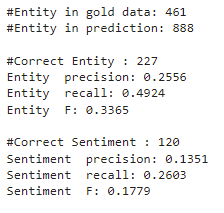
* Description (Initial stage):
  + Processes the transition from START state to the first state, stores only one sequence for layer 0 as it only has one path to the node.
* Description (Main body):
  + Processes the transition from state *yi* to state *yi+1* , for each node in the layer, run a helper function to return only the k-best paths from START to the current node.
* Description (Final stage):
  + Acquire the k-best sequence result and probability for that sequence from the last layer of the dictionary
* Input:
  + Input list, start\_p, trans\_p, emit\_p, tags\_dictionary, tags\_count, default k=5
* Output:
  + viterbi sequence dictionary, k-best probabilities, k-best sequences, kth-best probability, kth-best sequence

The function name *viterbi\_k()* is named as such as it can also return the *k-th* best sequence, so not necessarily only the 5th best. The k parameters can be changed in the default input parameters the function takes in.

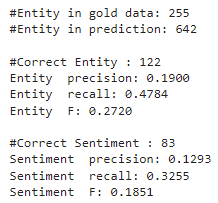
Since we store the k-best states for the sequences, we do not to go through a backtracking step to get the argmax for the k-best probabilities. This algorithm works because at every node in the Hidden Markov Model, the best k-paths are already determined up until that point. At the last layer we will have k \* len(tags) number of sequences, and what we can do from there is just find the k-best overall. This guarantees that we can find the kth-best sequence in the algorithm. The time complexity of the algorithm is . Where n is the length of the input, T is the number of possible tags, and k is the intended best sequence we want to find.

One thing to note is this algorithm is also done without accounting for underflow error, since we did not encounter any while testing. As such, we did not use the log probability to account for underflow.

## 3.3 Results



*Fig 3.3.1: Results for ./ES/dev.in*



*Fig 3.3.2: Results for ./RU/dev.in*

As predicted, for the 5-th best sequence we get an overall lower sentiment F-score as compared to the best algorithm method. This goes for both the RU and ES set.

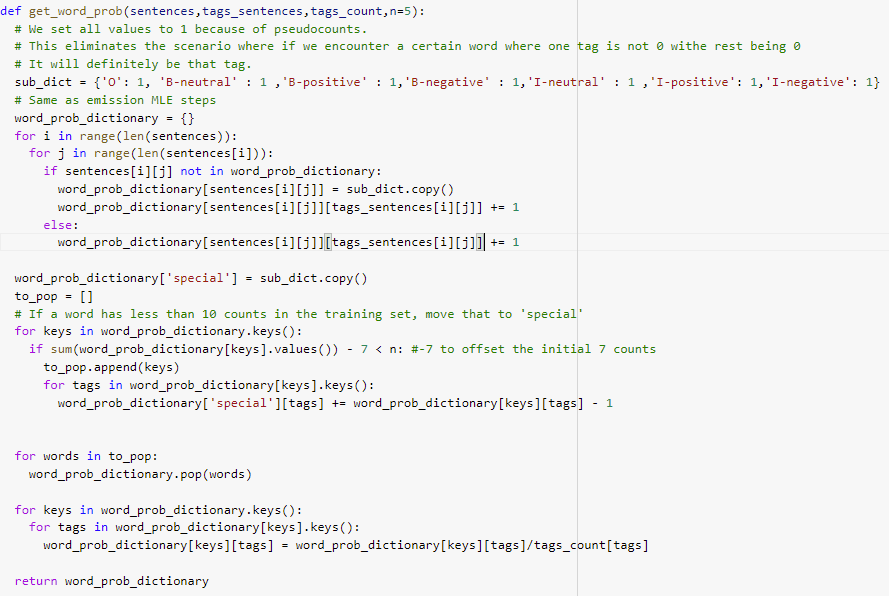
4. Question 4

For this question we did 4 approaches. This section we will cover the implementation and results for all of them. And conclude which one we intend to use for the held out test sets.

4.1 Naive Bayes

4.1.1 Preprocessing

In the Naive Bayes method, we calculated the conditional probability of the tag given x by the formula . To account for the unknown words, we set a rule whereby if we encounter a particular word less than *n* times in the training set, we categorise that word as *special* and then update it's tag probability from there. The corresponding function is as follows:



*Fig 4.1.1.1: get\_word\_prob() code block*

* Description:
  + Calculates the probability a word emits a certain tag, similar to emission probability calculation
* Input:
  + List of sentences, List of tags corresponding to the given sentence, total tags count, default value n = 5.
* Output:
  + Dictionary containing all the possible tags and probabilities the word can emit.



*Fig 4.1.1.2: example output for get\_word\_prob()*

This function also uses the pseudocounts method as mentioned in Section 3. This is to eliminate unfair classification.

We also defined a function to classify the possible tag probability:



*Fig 4.1.1.3: get\_tag\_prob() example*

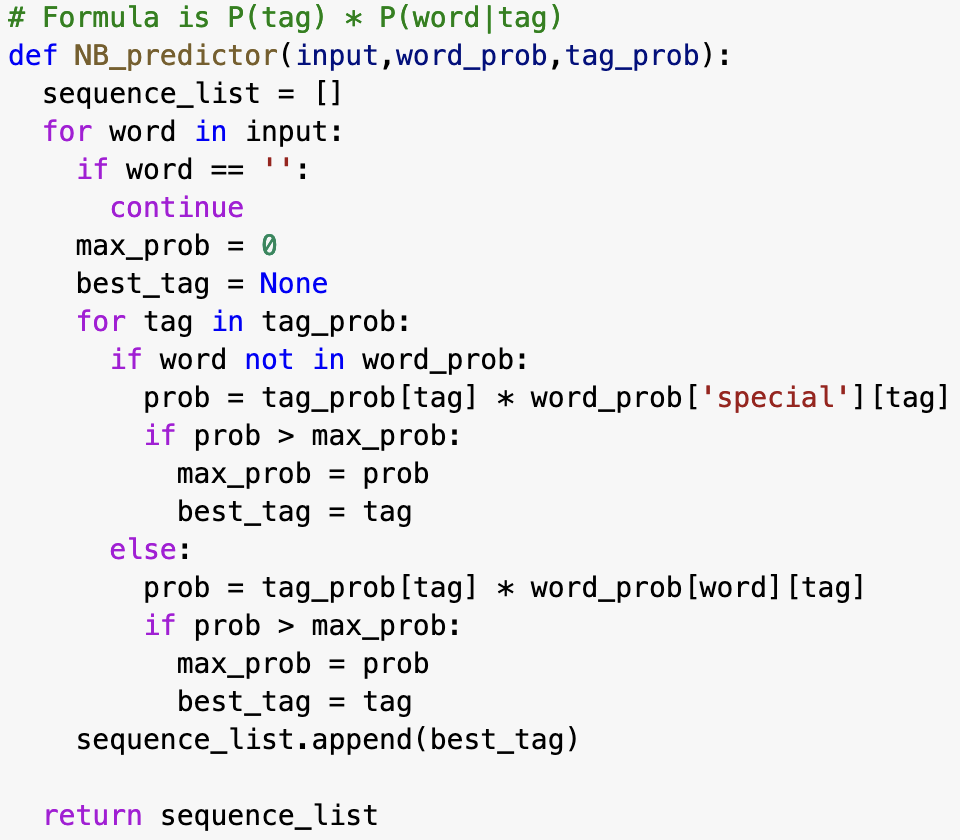
* Description:
  + Calculates the probability of the tag appearing overall.
* Input:
  + Tags count
* Output:
  + Dictionary containing the probabilities of



*Fig 4.1.1.4: example output for get\_tag\_prob()*

With these two functions, we can now execute the naive bayes predictor.

4.1.2 Main code

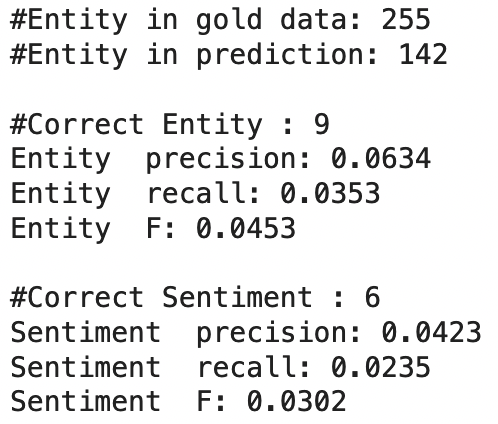


*Fig 4.1.2.1: NB\_predictor() code block*

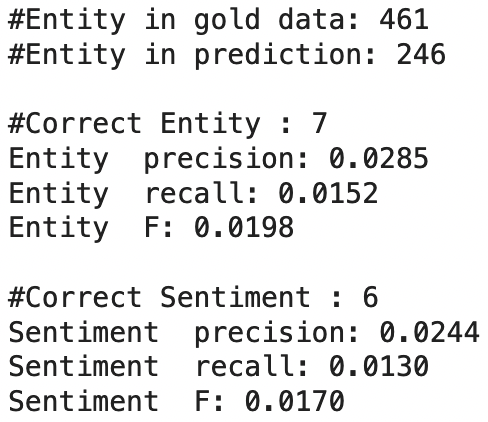
* Description:
  + Calculates the most likely sequence for a given sentence using Naive Bayes
* Input:
  + List of words [x], word\_probabilities, tag\_probabilities
* Output:
  + List of sequences [y]

This block shows the execution of Naive Bayes. For each word in the input, we run the formula . If we then encounter a word from the test set that is not in the training set, we will use the special token *special* to classify the word.

4.1.3 Results



*Fig 4.1.3.1: Results for ./ES/dev.in*

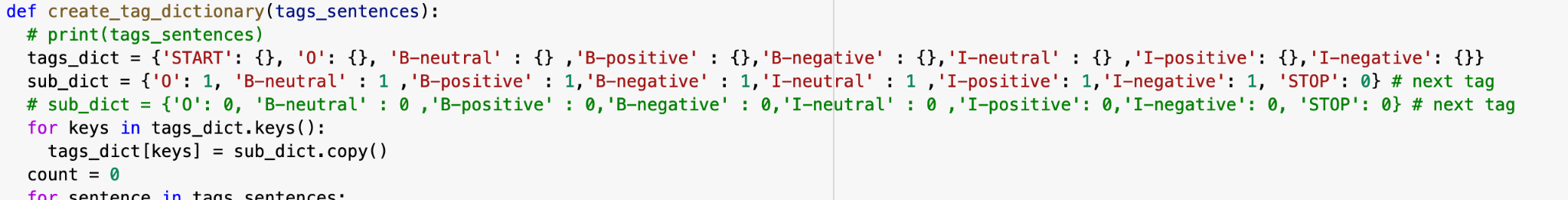


*Fig 4.1.3.2: Results for ./RU/dev.in*

The above figures show the results for the Naive Bayes method. The performance is rather poor, with only 6 correct sentiments for each test set. This is most likely due to being unable to group the words together in Naive Bayes. Since we only predict the likely tag for that particular word. We do not consider the output of the previous word or the word that comes after it. This could also be due to using the *special* token, which in essence just considers only the highest probability tag for those lesser appearing words.

4.2 HMM with pseudocounts

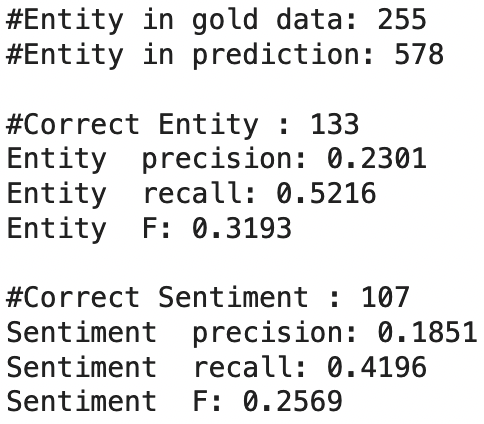
4.2.1 Preprocessing



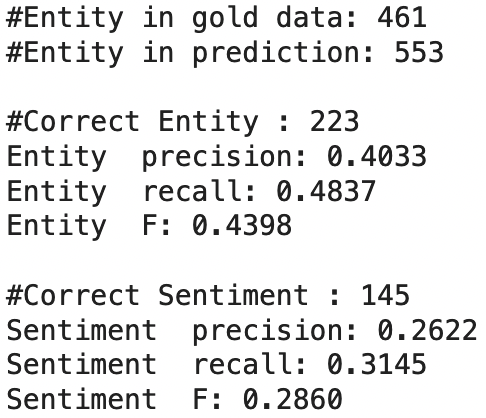
*Fig 4.2.1.1: create\_tag\_dictionary() pseudocount*

Similar to question 3, we updated the code for *create\_tag\_dictionary()* to include pseudocounts. This then eliminates the need for an alternative solution for those None values that we encountered.

4.2.2 Results



*Fig 4.2.2.1: Results for ./ES/dev.in*



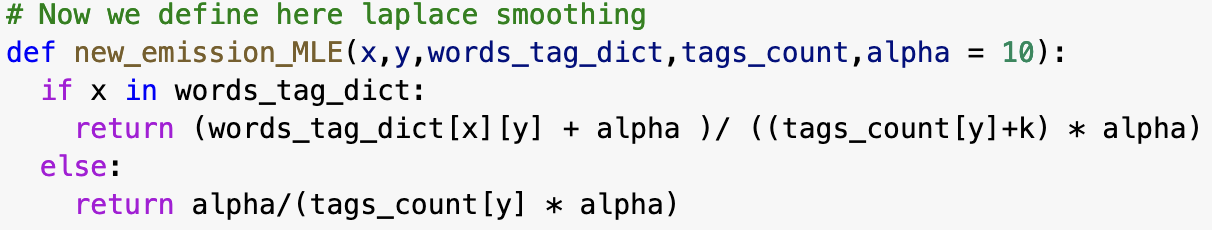
*Fig 4.2.2.2: Results for ./RU/dev.in*

The results here are almost similar to the ones we have for question 2. With only a slight variation in recall. However, this method allows for a simpler approach to the None issue we encountered previously.

4.3 HMM with laplace smoothing and pseudocounts

4.3.1 Preprocessing

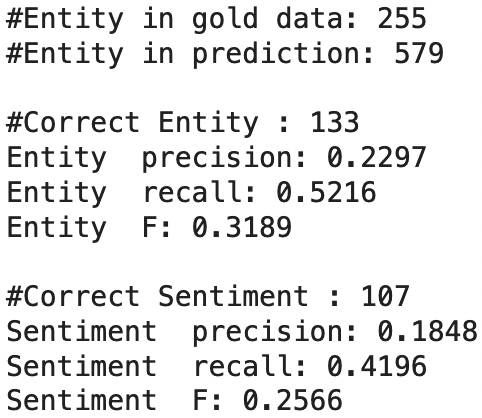
Laplace smoothing is another method to handle the emission MLE of unseen words. It works similar to the previous k method, but this time we multiply the alpha value instead of adding it.



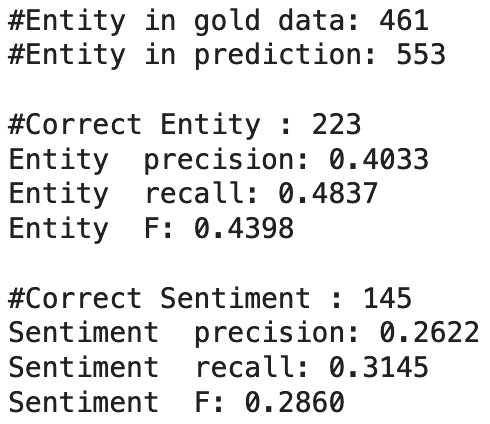
*Fig 4.3.1.1: new\_emission\_MLE() laplace smoothing*

This way, we can justify how much weight we want to put on the unknown words.

4.3.2 Results



*Fig 4.3.2.1: Results for ./ES/dev.in*

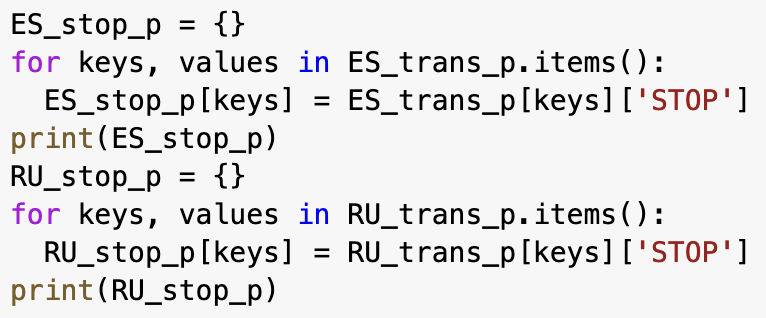


*Fig 4.3.2.2: Results for ./RU/dev.in*

This is the result for the optimal value when alpha = 10. Even so there is almost no change in the results for this and the previous method.

4.4 HMM with soft EM and laplace smoothing and pseudocounts

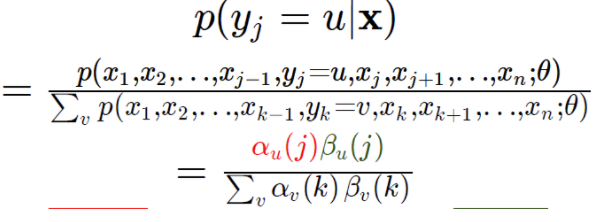
4.4.1 Preprocessing



*Fig 4.4.1.1: Creating stop\_p dictionaries*

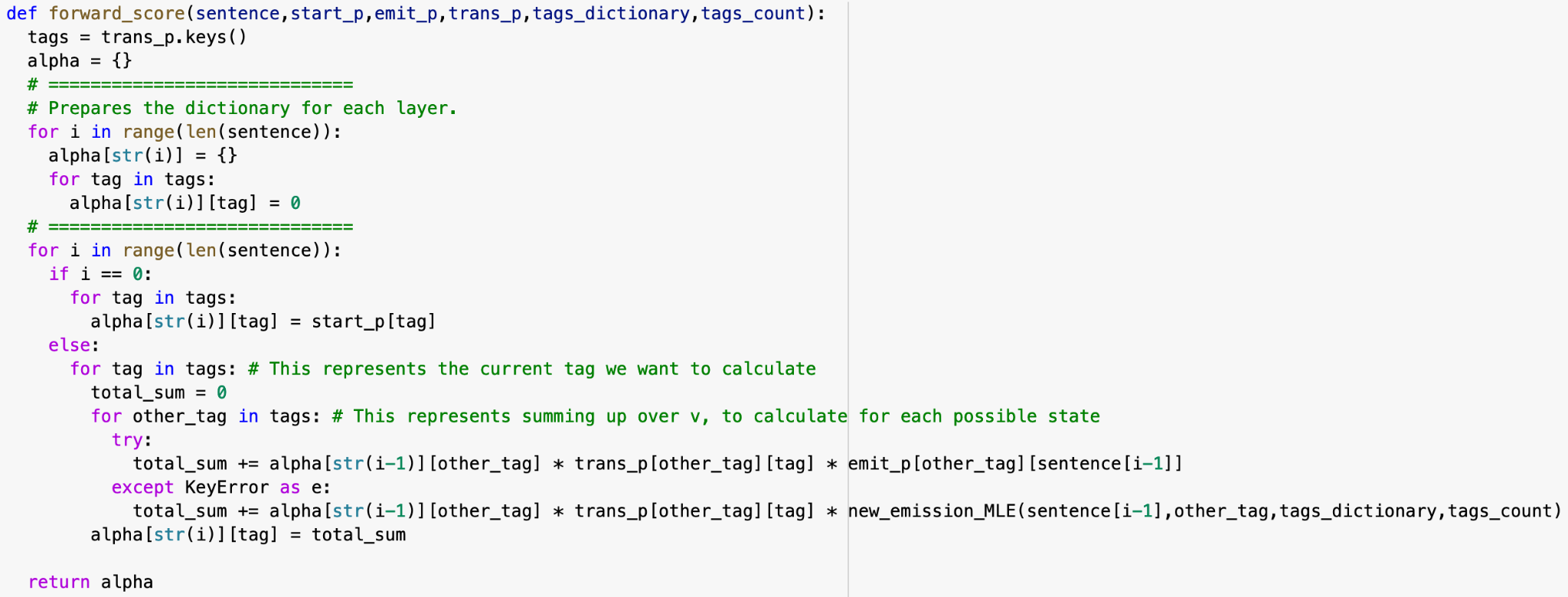
The last method we tried was using soft EM instead of a hard EM method. To do this, we needed to create a stop dictionary to be used in backwards calculation. This is simple to do, we just took the items in the transition probabilities and assigned the stop probabilities for the respective datasets.

4.4.2 Soft EM



*Fig 4.4.2.1: Probability formula for soft EM*

For this approach, we used this formula to calculate the probabilities of given tags. We would only consider the numerators in as the denominators for each tag in the same node has the same value.

**

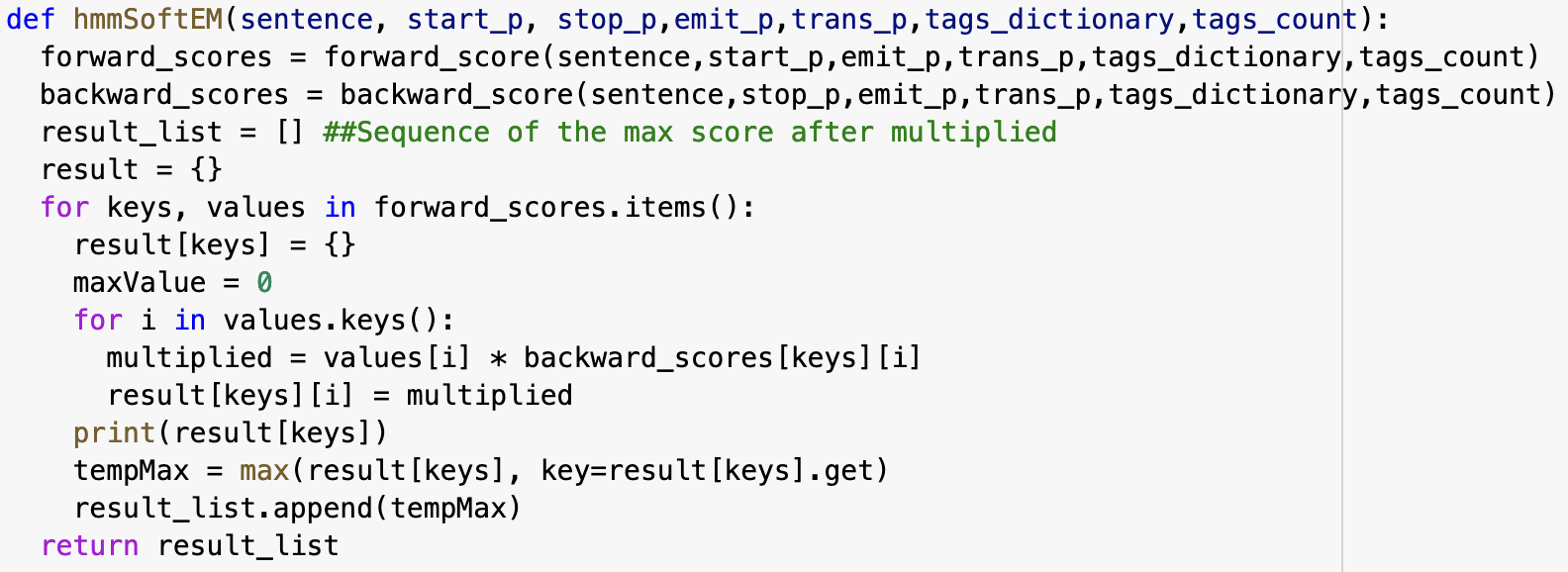
*Fig 4.4.2.1: forward\_score() code block*

* Description:
  + Calculates all forward scores for each word in the sentence
* Input:
  + List of words [x], start probabilities, emission probabilities, transmission probabilities, tags dictionary, tags count.
* Output:
  + Dictionary containing all the forward scores for each tag for the word.

**

*Fig 4.4.2.2: backward\_score() code block*

* Description:
  + Calculates all backwards scores for each word in the sentence
* Input:
  + List of words [x], stop probabilities, emission probabilities, transmission probabilities, tags dictionary, tags count.
* Output:
  + Dictionary containing all the backwards scores for each tag for the word.

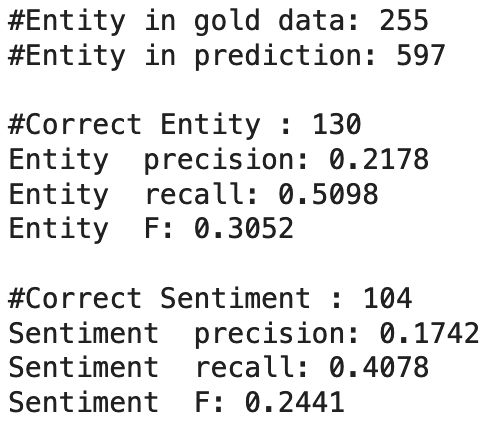
**

*Fig 4.4.2.3: hmmSoftEM() code block*

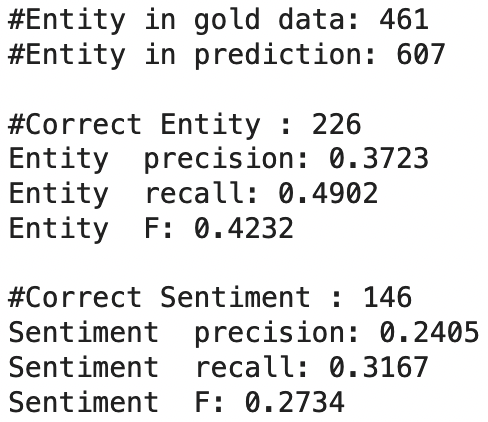
* Description:
  + Gets the best sequence by applying the formula to calculate the probability each word produces the given tag.
* Input:
  + List of words [x], start probabilities, stop probabilities, emission probabilities, transmission probabilities, tags dictionary, tags count.
* Output:
  + List of tag sequences [y]

In this approach, we calculated the forward and backwards scores for the sentences. After that we multiplied to check which probability has the highest value, we then get the *argmax* of that probability to find the most likely tag that the word will give.

4.4.3 Results



*Fig 4.4.3.1: Results for ./ES/dev.in*



*Fig 4.4.3.2: Results for ./ES/dev.in*

For this method, we encountered more predicted entities as compared to using hard EM. Although we did perform differently for different datasets. For the ES set, we performed slightly worse, predicted lesser correct entities and sentiments. However for the RU set, we performed slightly better and that gave us more desirable results.

4.5 Conclusion

Overall through all these methods, we have decided to use the method of using soft EM to predict the held out test sets. This method should provide us with a clearer understanding of HMM as we tried in both the soft EM and hard EM methods.

4.6 References

1. Naive Bayes, Wikipedia: <https://en.wikipedia.org/wiki/Naive_Bayes_classifier>
2. Hidden Markov Model, Wikipedia: <https://en.wikipedia.org/wiki/Hidden_Markov_model>