50.007 Machine Learning, Summer 2023 Design Project

Team Members:
Vanya Jalan (1006190)
Rishika Banka (1006304)
Jignesh Motwani (1006178)

PART 1

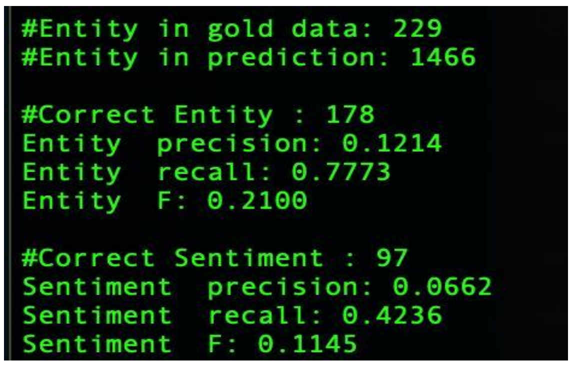
Code Execution:
Simply run the file part1_code.py, that will generate the required output files for both ES and RU datasets, in their respective folders.

Code Explanation:
We first estimate the emission parameters from the training data. The parameters are used to calculate the probabilities of each token (word) given a specific tag (sentiment or part-of-speech label). The function takes the path to the training file and an optional smoothing parameter k (default value is 1).
This is how the code works:
1. Initialization: The function initializes two defaultdicts, `token_tag_count` and `tag_count`, to count the frequency of token-tag pairs and tags, respectively.
2. Reading Training Data: The function reads the training data line by line, splitting each line into a token and its corresponding tag.
3. Counting Frequencies: It updates the counts for each token-tag pair and each tag.
4. Calculating Emission Parameters: The function calculates the emission parameters using the counts, applying smoothing with parameter k. It also handles unknown tokens by associating them with a special `'#UNK#'` token.
5. Return Value: The function returns the calculated emission parameters as a defaultdict of defaultdicts.

We then perform sentiment analysis on a given input file using the emission parameters and write the results to the output file.
This is how the code for it works:
1. Initialization: The function initializes an empty list, `output_lines`, to store the output lines.
2. Reading Input Data: It reads the input data line by line, treating each non-empty line as a token.
3. Predicting Tags: If a token is not found in the emission parameters, it is replaced with the special `'#UNK#'` token. For each token, the function predicts the tag with the maximum emission probability.
4. Writing to Output File: It finally writes the token and the predicted tag to the output file in the desired format.

Here is a screenshot of the Precision, Recall and F-values we get on running the evalScript provided to us:

ES

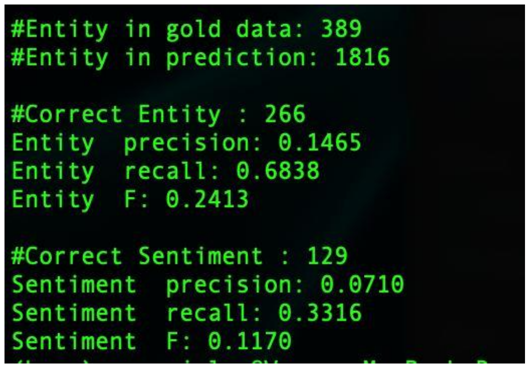


```
#Entity in gold data: 229
#Entity in prediction: 1466

#Correct Entity : 178
Entity  precision: 0.1214
Entity  recall: 0.7773
Entity  F: 0.2100

#Correct Sentiment : 97
Sentiment  precision: 0.0662
Sentiment  recall: 0.4236
Sentiment  F: 0.1145
```

RU



```
#Entity in gold data: 389
#Entity in prediction: 1816

#Correct Entity : 266
Entity  precision: 0.1465
Entity  recall: 0.6838
Entity  F: 0.2413

#Correct Sentiment : 129
Sentiment  precision: 0.0710
Sentiment  recall: 0.3316
Sentiment  F: 0.1170
```

PART 2

Code Execution:
Simply run the file part2_code.py, that will generate the required output files for both ES and RU datasets, in their respective folders.

Code Explanation:
We implemented the Viterbi Algorithm here.

For the emission parameters, we used the same code as we had in Part 1.

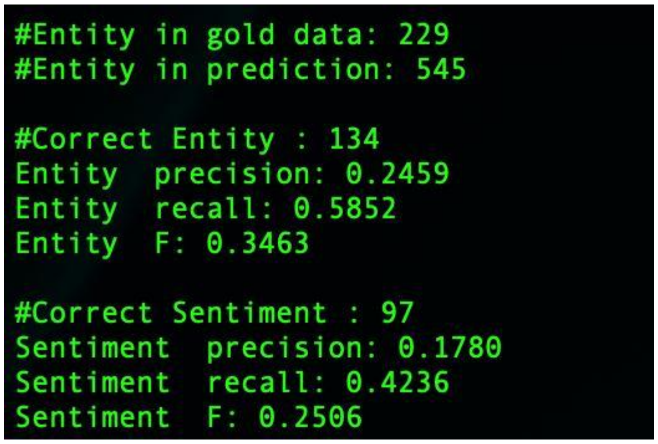
For transition parameters, we defined a new function, by using the following steps:
1. <u>Initialize Counters</u>: Two dictionaries (`transition_count` and `tag_count`) are used to keep track of the counts of transitions between tags and the individual tag counts.
2. <u>Read the File</u>: The code reads the training file line by line, splitting each line into a word and tag. Empty lines are used to indicate the end of a sentence.
3. <u>Update Counts</u>: For each line, the code updates the counts of transitions from the previous tag to the current tag, and increments the count for the previous tag.
4. <u>Handle Sentence Boundaries</u>: If an empty line is encountered (indicating the end of a sentence), the transition to the `STOP` tag is counted, and the `START` tag is set as the previous tag for the next sentence.
5. <u>Calculate Probabilities</u>: Finally, the code calculates the transition probabilities by dividing the transition counts by the corresponding tag counts.

We now broke down the dev.in test input file to be processed as sentences, every empty line was considered the end of a sentence and we passed these sentences into our Viterbi Algorithm function that labels the data according to the emission and transition parameters.
This is how the code for it works:
1. <u>Initialization:</u> We initialize a matrix to store the probabilities for each state at each step. It's implemented as a dictionary of dictionaries. We also initialize a structure to store backpointers to keep track of the previous state for each state at each step and set the initial probability of the 'START' state to 1.0.
2. <u>Get the next states:</u> We then iterate through the words in the sentence. If the word has never been seen before in the train data, we set the word to “#UNK”. We now iterate through the emission_parameters dictionary for our word to get the next possible states for our current word.
3. <u>Calculate score for each next state</u>: We then calculate the score for each of these next states as the score for our current state * transition probability from current state to next state * emission probability of the next state, and store the highest score out of all these in our Viterbi matrix. We also update the backpointer dictionary to store the backpointer i.e. previous state corresponding to our current state.
4. <u>Final Path Reconstruction</u>: We find the last tag that has the maximum score and construct the path using the backpointer dictionary. We then reverse and return this path as the optimal sequence for our input sentence. (if such a path does not exist, we return a default path with all tags as ‘O’)

The output is then formatted again to be printed to the output files as stated in the question.

Here is a screenshot of the Precision, Recall and F-values we get on running the evalScript provided to us:

ES

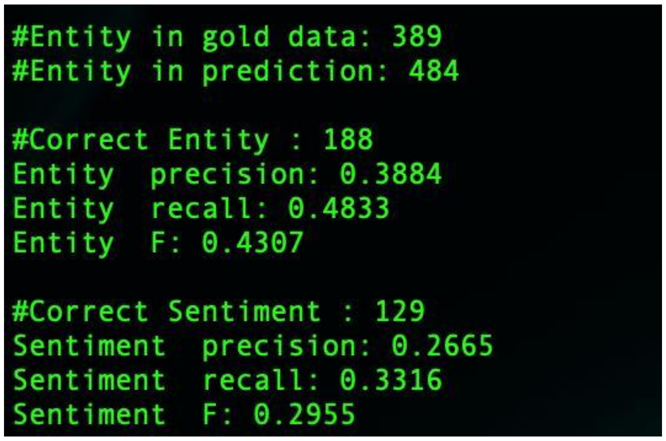


```
#Entity in gold data: 229
#Entity in prediction: 545

#Correct Entity : 134
Entity  precision: 0.2459
Entity  recall: 0.5852
Entity  F: 0.3463

#Correct Sentiment : 97
Sentiment  precision: 0.1780
Sentiment  recall: 0.4236
Sentiment  F: 0.2506
```

RU



```
#Entity in gold data: 389
#Entity in prediction: 484

#Correct Entity : 188
Entity  precision: 0.3884
Entity  recall: 0.4833
Entity  F: 0.4307

#Correct Sentiment : 129
Sentiment  precision: 0.2665
Sentiment  recall: 0.3316
Sentiment  F: 0.2955
```

PART 3

Code Execution:
Simply run the file part3_code.py, that will generate the required output files for both ES and RU datasets, in their respective folders.

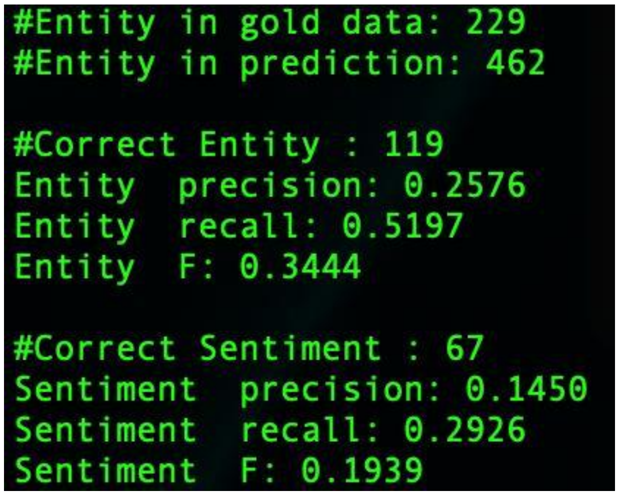
Code Explanation:
We modified the Viterbi Algorithm here, to give the kth best sequence. We modified the Viterbi Algorithm from part 2 here. The emission parameters and transition parameters are the same as used in the previous parts.

Here is how the modified function works:
1. Initialization: The Viterbi matrix is initialized using a default dictionary, where each cell contains a list of tuples. Each tuple represents one of the top k paths and contains a probability score and the corresponding path. The start state is initialized with a probability of 1.0.
2. Calculate paths: This code checks if the current word is in the emission parameters. If not, it treats the word as an unknown word (`#UNK#`).
Then a nested loop calculates the top k paths for each tag `v` for the current word, based on the top k paths for each tag `u` for the previous word. The new score is calculated by multiplying the score of the previous path with the transition probability from `u` to `v` and the emission probability of the current word given `v`. The new paths are stored in the Viterbi matrix.
3. Return kth best path: We then collect all the paths at the end of the sentence and sort them by score, returning the kth best path.
(If there are fewer than k paths, it returns a default path with all 'O' tags.)

The output again is then formatted again to be printed to the output files as stated in the question.

Here is a screenshot of the Precision, Recall and F-values we get on running the evalScript provided to us:

ES, k=2

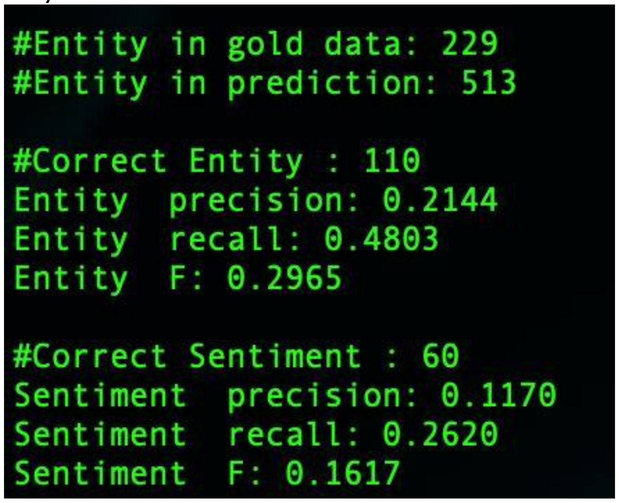


```
#Entity in gold data: 229
#Entity in prediction: 462

#Correct Entity : 119
Entity  precision: 0.2576
Entity  recall: 0.5197
Entity  F: 0.3444

#Correct Sentiment : 67
Sentiment  precision: 0.1450
Sentiment  recall: 0.2926
Sentiment  F: 0.1939
```

ES, k=8

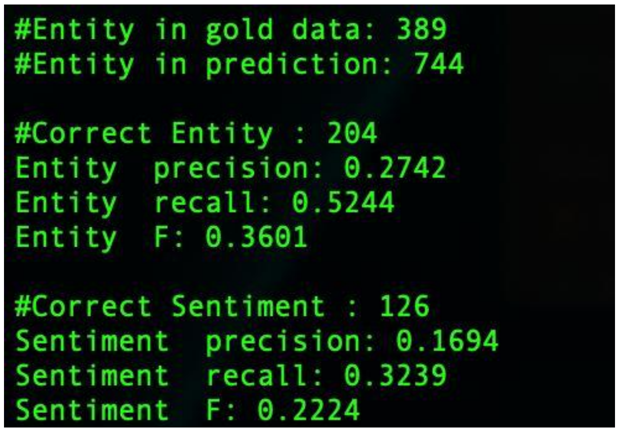


```
#Entity in gold data: 229
#Entity in prediction: 513

#Correct Entity : 110
Entity  precision: 0.2144
Entity  recall: 0.4803
Entity  F: 0.2965

#Correct Sentiment : 60
Sentiment  precision: 0.1170
Sentiment  recall: 0.2620
Sentiment  F: 0.1617
```

RU, k=2

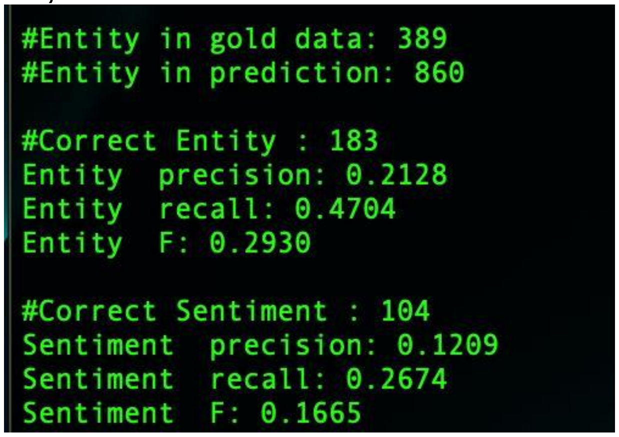


```
#Entity in gold data: 389
#Entity in prediction: 744

#Correct Entity : 204
Entity  precision: 0.2742
Entity  recall: 0.5244
Entity  F: 0.3601

#Correct Sentiment : 126
Sentiment  precision: 0.1694
Sentiment  recall: 0.3239
Sentiment  F: 0.2224
```

RU, k=8



```
#Entity in gold data: 389
#Entity in prediction: 860

#Correct Entity : 183
Entity  precision: 0.2128
Entity  recall: 0.4704
Entity  F: 0.2930

#Correct Sentiment : 104
Sentiment  precision: 0.1209
Sentiment  recall: 0.2674
Sentiment  F: 0.1665
```

PART 4

Code Execution:

Simply run the file part4_code.py, that will generate the required output files for both ES and RU datasets, as well as the test data outputs in their respective folders with the file names specified in the question

Code Explanation:

The code implements a trigram, third-order Hidden Markov Model (HMM) with the Viterbi algorithm.

1. Initialization: Inside the class, the `__init__` method initializes various attributes such as words, tag-word counts, transmissions, and other variables that will be used throughout the implementation.
2. Reading File: The method `read_file` reads the training data file and processes the data into suitable structures for training.
3. Training Process: The model learns from the training data, calculating the probabilities for transitions and emissions.

- ***Emission Probabilities:***
  1. tag_word_count Dictionary: This dictionary holds the counts of each tag-word combination.
  2. Normalization: The counts are normalized by dividing by the total count of the corresponding tag.
  3. Matrix Formation: The probabilities are stored in a matrix for efficient access.

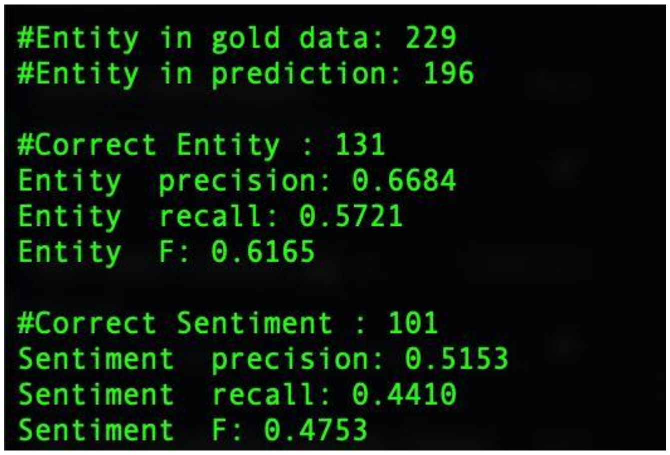
- ***Transition Probabilities:***
  1. Transition Matrices: Transition matrices are created to store the probabilities of transitions between different tags.
  2. Transition Counting: Transition probabilities are populated into the matrices using counts from a dictionary like \`uvw_dic\` that stores the transition probabilities keeping in account the two previous tags (trigram approach). The matrices are then converted into panda dataframes for easier manipulation.

4. Viterbi Algorithm Implementation: A method within the class implements the Viterbi algorithm to find the most probable sequence of hidden states for a given observation sequence. The dynamic programming table used in Viterbi Algorithm considers the two previous tags when calculating the probability of the current tag.

The output again is then formatted again to be printed to the output files as stated in the question.

Here is a screenshot of the Precision, Recall and F-values we get on running the evalScript provided to us:

ES

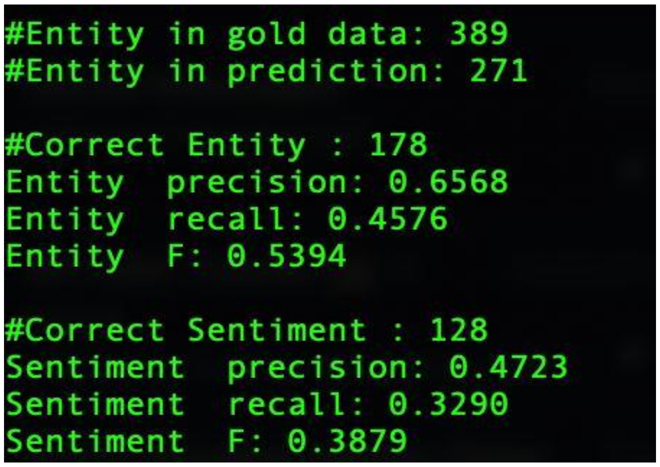


```
#Entity in gold data: 229
#Entity in prediction: 196

#Correct Entity : 131
Entity  precision: 0.6684
Entity  recall: 0.5721
Entity  F: 0.6165

#Correct Sentiment : 101
Sentiment  precision: 0.5153
Sentiment  recall: 0.4410
Sentiment  F: 0.4753
```

RU



```
#Entity in gold data: 389
#Entity in prediction: 271

#Correct Entity : 178
Entity  precision: 0.6568
Entity  recall: 0.4576
Entity  F: 0.5394

#Correct Sentiment : 128
Sentiment  precision: 0.4723
Sentiment  recall: 0.3290
Sentiment  F: 0.3879
```

The test data files have also been generated and saved under the ES and RU folders, with the name as mentioned in the question.

NOTE:
All the output files can be found inside the ES and RU folders according to the naming convention specified in the question.
(The ES and RU folders are inside the Data folder)

Please do not change any of the directory structure as the train, test data files are given a specific path in the code, so if changed the code will not be able to find the said file and give errors.