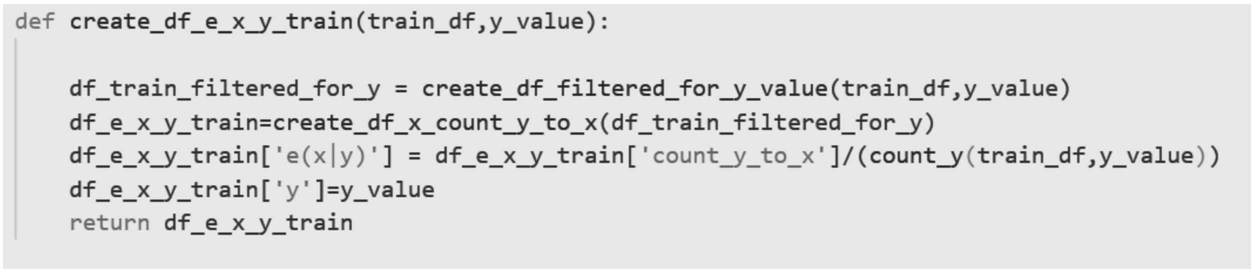
ML Project

Sherin Karuvallil Saji 1005228
Venkatakrishnan Logganesh 1006050
Mehta Yash Piyush 1006516
Tang Heng 1006102

Part 1

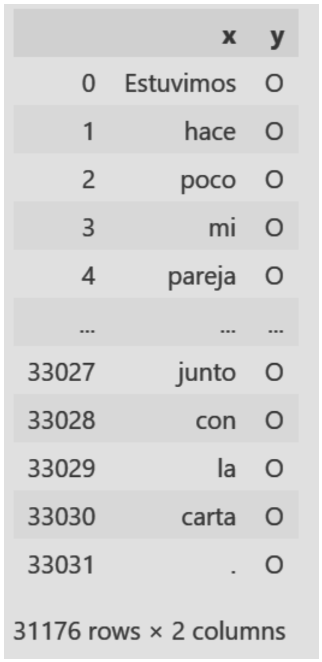
Question 1



```
def create_df_e_x_y_train(train_df,y_value):

    df_train_filtered_for_y = create_df_filtered_for_y_value(train_df,y_value)
    df_e_x_y_train=create_df_x_count_y_to_x(df_train_filtered_for_y)
    df_e_x_y_train['e(x|y)'] = df_e_x_y_train['count_y_to_x']/(count_y(train_df,y_value))
    df_e_x_y_train['y']=y_value
    return df_e_x_y_train
```

This function takes in a dataframe containing training set data and a y_value. It creates a new dataframe called df_train_filtered_for_y by filtering the inputted train_df for the y_value. Now, df_train_filtered_for_y contains all the rows of the df_train that has y=y_value. Then, we create df_e_x_y_train that maps each x value that appears in df_train_filtered_for_y to the number of times it appears in df_train_filtered_for_y. These counts are in column count_y_to_x that represents Count (y →x). Then, we create another column called ‘e(x|y)’ in df_train and this column will contain the e(x|y) values which is calculated by taking count_y_to_x / number of times the y value appears in the train_df. Then, the function returns df_e_x_y_train.

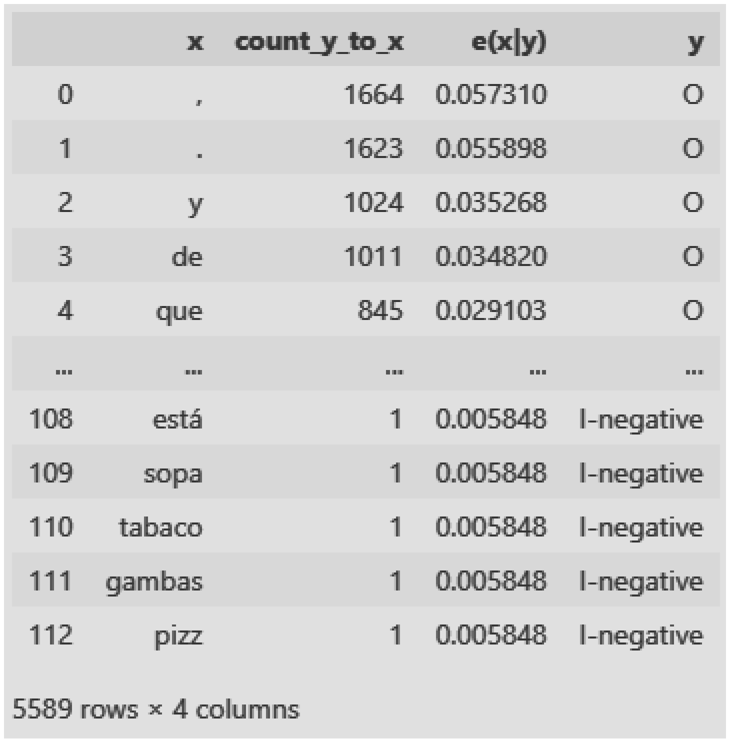
Here is the train_df made from spanish train dataset.



	x	y
0	Estuvimos	O
1	hace	O
2	poco	O
3	mi	O
4	pareja	O
...	...	...
33027	junto	O
33028	con	O
33029	la	O
33030	carta	O
33031	.	O

31176 rows × 2 columns

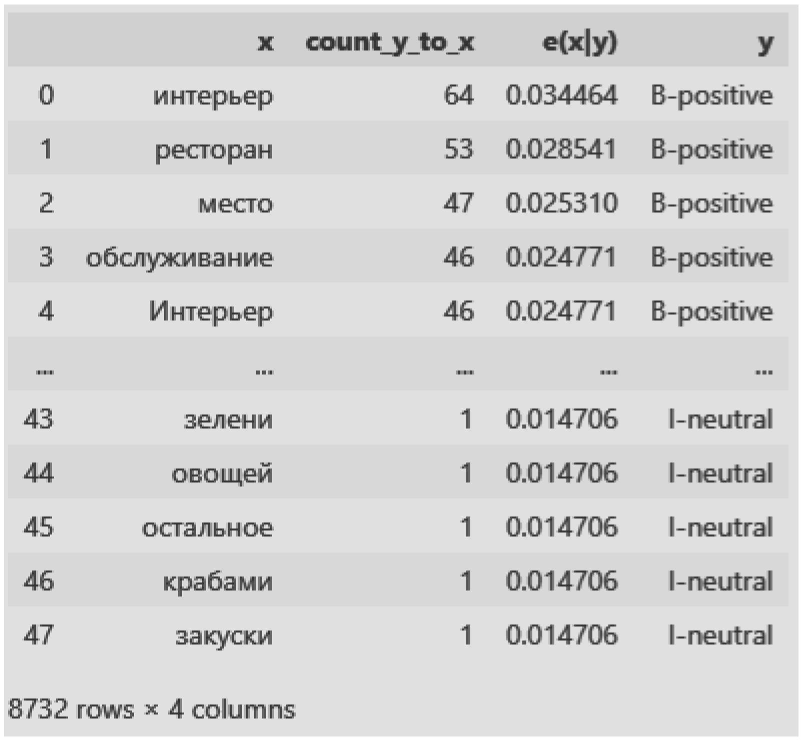
We run this function for all y values and join the outputted dataframes together.
Here is what the joint dataframe for spanish looks like:



	x	count_y_to_x	e(x|y)	y
0	,	1664	0.057310	O
1	.	1623	0.055898	O
2	y	1024	0.035268	O
3	de	1011	0.034820	O
4	que	845	0.029103	O
...	...	...	...	...
108	está	1	0.005848	I-negative
109	sopa	1	0.005848	I-negative
110	tabaco	1	0.005848	I-negative
111	gambas	1	0.005848	I-negative
112	pizz	1	0.005848	I-negative

5589 rows × 4 columns

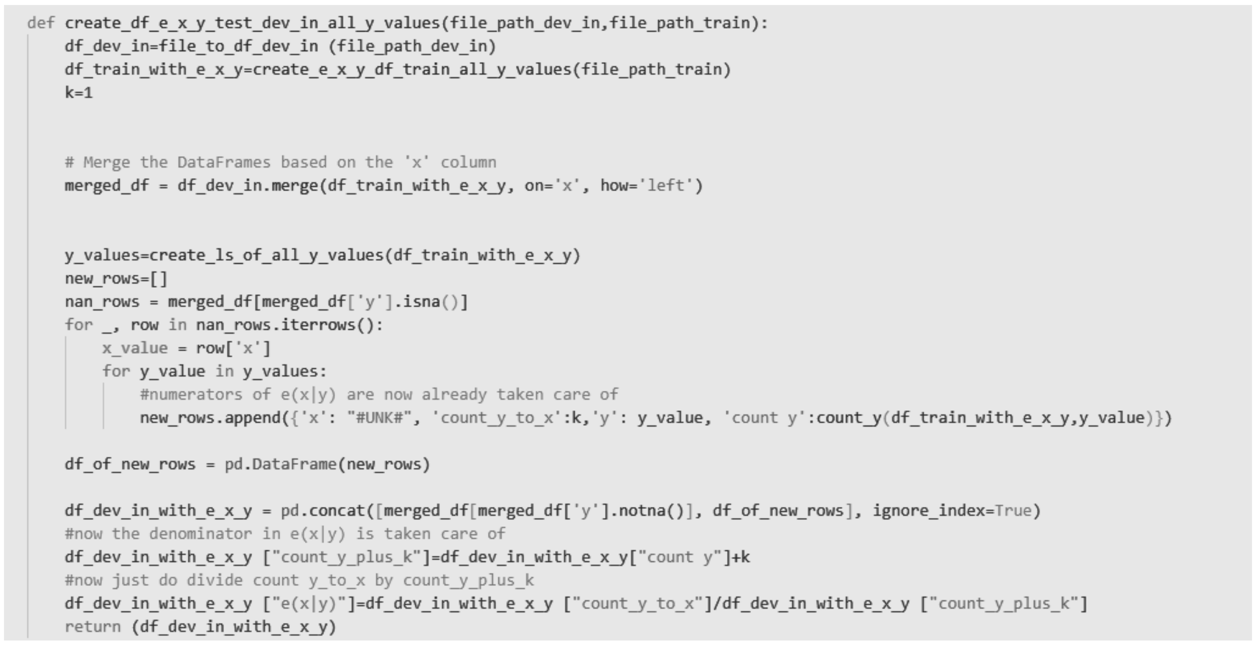
Here is what the joint dataframe for russian looks like:



	x	count_y_to_x	e(x|y)	y
0	интерьер	64	0.034464	B-positive
1	ресторан	53	0.028541	B-positive
2	место	47	0.025310	B-positive
3	обслуживание	46	0.024771	B-positive
4	Интерьер	46	0.024771	B-positive
...	...	...	...	...
43	зелени	1	0.014706	I-neutral
44	овощей	1	0.014706	I-neutral
45	остальное	1	0.014706	I-neutral
46	крабами	1	0.014706	I-neutral
47	закуски	1	0.014706	I-neutral

8732 rows × 4 columns

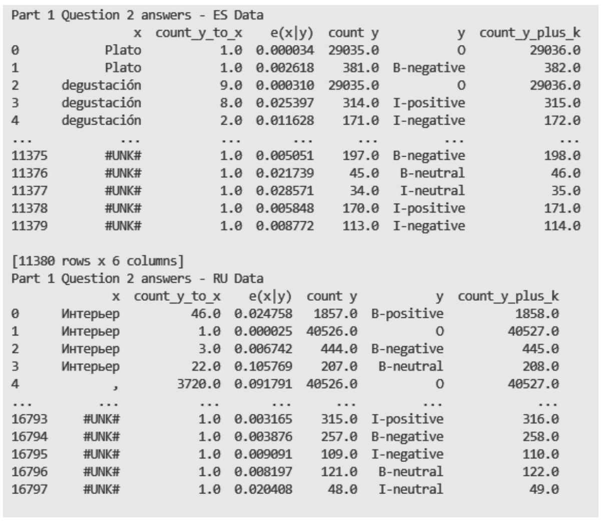
Question 2



```
def create_df_e_x_y_test_dev_in_all_y_values(file_path_dev_in,file_path_train):
    df_dev_in=file_to_df_dev_in (file_path_dev_in)
    df_train_with_e_x_y=create_e_x_y_df_train_all_y_values(file_path_train)
    k=1


    # Merge the DataFrames based on the 'x' column
    merged_df = df_dev_in.merge(df_train_with_e_x_y, on='x', how='left')


    y_values=create_ls_of_all_y_values(df_train_with_e_x_y)
    new_rows=[]
    nan_rows = merged_df[merged_df['y'].isna()]
    for _, row in nan_rows.iterrows():
        x_value = row['x']
        for y_value in y_values:
            #numerators of e(x|y) are now already taken care of
            new_rows.append({'x': "#UNK#", 'count_y_to_x':k,'y': y_value, 'count y':count_y(df_train_with_e_x_y,y_value)})

    df_of_new_rows = pd.DataFrame(new_rows)

    df_dev_in_with_e_x_y = pd.concat([merged_df[merged_df['y'].notna()], df_of_new_rows], ignore_index=True)
    #now the denominator in e(x|y) is taken care of
    df_dev_in_with_e_x_y ["count_y_plus_k"]=df_dev_in_with_e_x_y["count y"]+k
    #now just do divide count y_to_x by count_y_plus_k
    df_dev_in_with_e_x_y ["e(x|y)"]=df_dev_in_with_e_x_y ["count_y_to_x"]/df_dev_in_with_e_x_y ["count_y_plus_k"]
    return (df_dev_in_with_e_x_y)
```

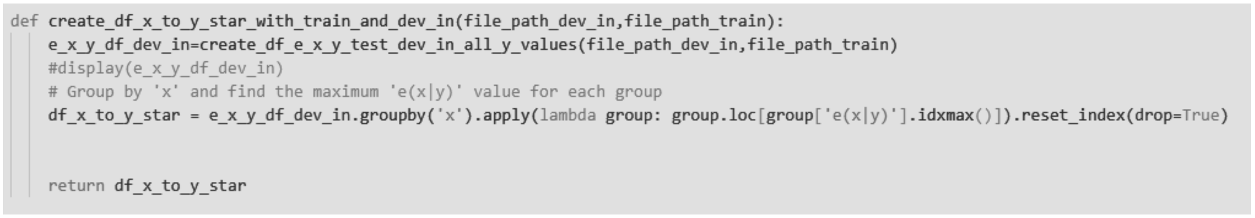
This function takes in a dev in dataset file path and a train dataset file path. It creates a dataframe from dev in dataset and df_train_with_e_x_y. We set k=1. Then,we merge df_dev_in and df_train_with_e_x_y together based on rows that have the same x values. Then, for every row in the dataframe that has a “NaN” value, these are the rows with x values that are not seen in the training set. These rows will be replaced with N other rows for each y label in the N y labels inside the df_train_with_e_x_y. We also set the count_y_to_x value of those rows to be k, taking care of the numerator. Then form the column count_y_plus_k and then we overwrite existing e(x|y) values with the new ones(it used to be from df_train_with_e_x_y because of the merger). The count y values and the count_y_to_x values were taken from df_train_with_e_x_y due to the merger.

This is what the dataframes will look like:



```
Part 1 Question 2 answers - ES Data
                x  count_y_to_x   e(x|y)  count y           y  count_y_plus_k
0           Plato           1.0  0.000034  29035.0          O         29036.0
1           Plato           1.0  0.002618    381.0  B-negative          382.0
2     degustación           9.0  0.000310  29035.0          O         29036.0
3     degustación           8.0  0.025397    314.0  I-positive          315.0
4     degustación           2.0  0.011628    171.0  I-negative          172.0
...           ...           ...       ...      ...         ...            ...
11375       #UNK#           1.0  0.005051    197.0  B-negative          198.0
11376       #UNK#           1.0  0.021739     45.0   B-neutral           46.0
11377       #UNK#           1.0  0.028571     34.0   I-neutral           35.0
11378       #UNK#           1.0  0.005848    170.0  I-positive          171.0
11379       #UNK#           1.0  0.008772    113.0  I-negative          114.0

[11380 rows x 6 columns]
Part 1 Question 2 answers - RU Data
             x  count_y_to_x   e(x|y)  count y           y  count_y_plus_k
0      Интерьер          46.0  0.024758   1857.0  B-positive         1858.0
1      Интерьер           1.0  0.000025  40526.0           O        40527.0
2      Интерьер           3.0  0.006742    444.0  B-negative          445.0
3      Интерьер          22.0  0.105769    207.0   B-neutral          208.0
4             ,        3720.0  0.091791  40526.0           O        40527.0
...         ...           ...       ...      ...         ...            ...
16793     #UNK#           1.0  0.003165    315.0  I-positive          316.0
16794     #UNK#           1.0  0.003876    257.0  B-negative          258.0
16795     #UNK#           1.0  0.009091    109.0  I-negative          110.0
16796     #UNK#           1.0  0.008197    121.0   B-neutral          122.0
16797     #UNK#           1.0  0.020408     48.0   I-neutral           49.0
```

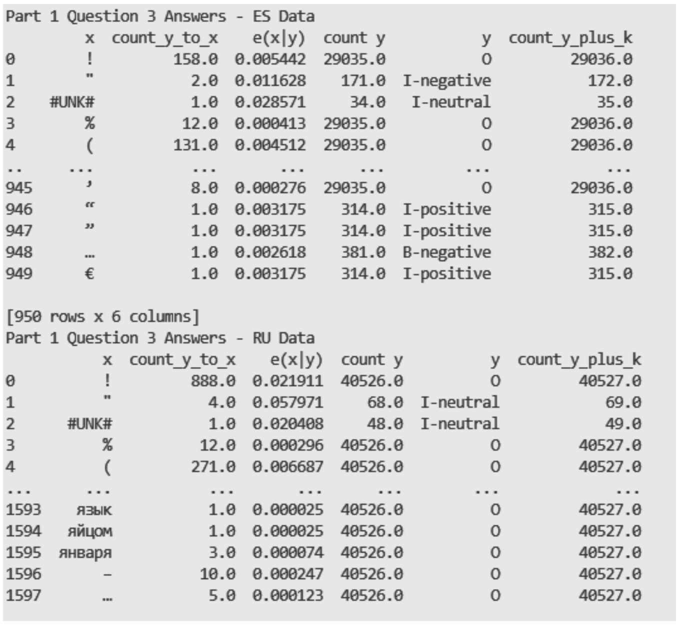
Question 3



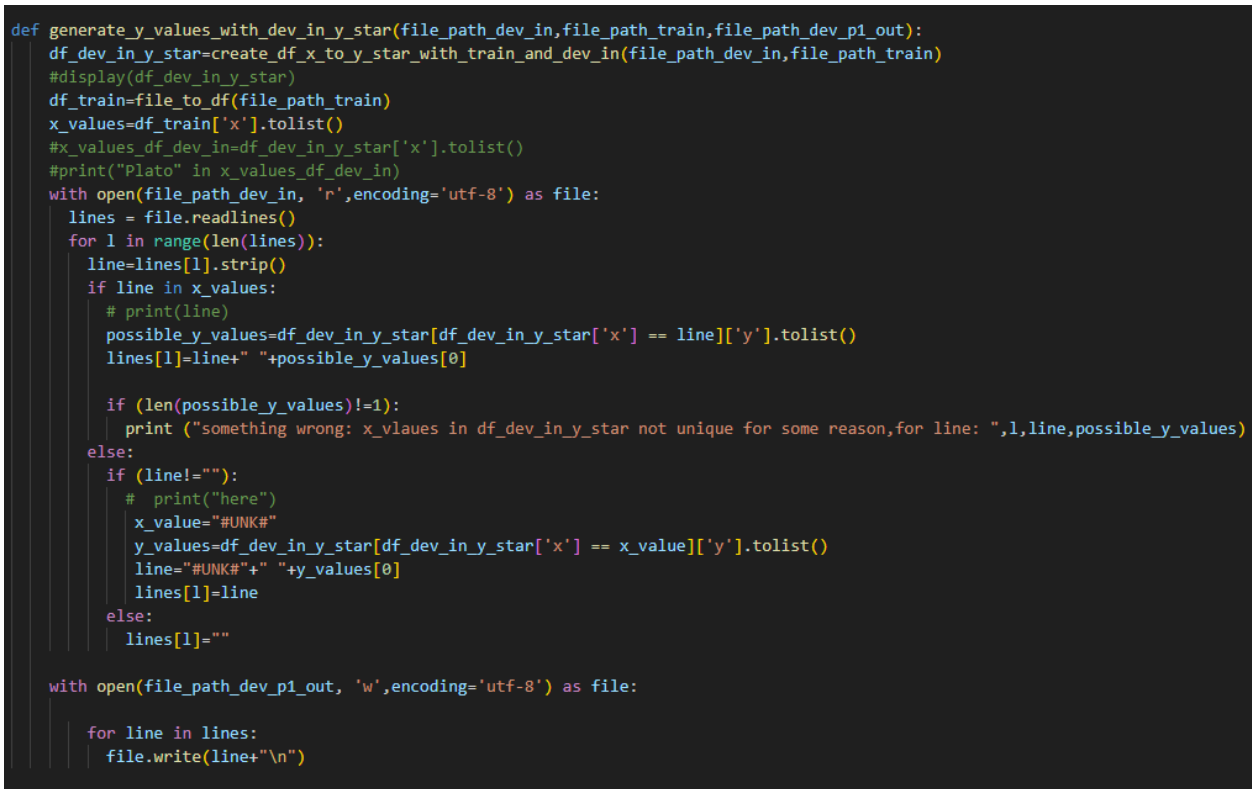
```python
def create_df_x_to_y_star_with_train_and_dev_in(file_path_dev_in,file_path_train):
    e_x_y_df_dev_in=create_df_e_x_y_test_dev_in_all_y_values(file_path_dev_in,file_path_train)
    #display(e_x_y_df_dev_in)
    # Group by 'x' and find the maximum 'e(x|y)' value for each group
    df_x_to_y_star = e_x_y_df_dev_in.groupby('x').apply(lambda group: group.loc[group['e(x|y)'].idxmax()]).reset_index(drop=True)


    return df_x_to_y_star
```

With this function we just group all the rows by their x values, then select the row that has the max e(x|y) of them for each of the groups of rows. This will give us df_x_to_y_star.

This is what the df_x_to_y_star looks like where the y column contains the y star and e(x|y) contains the maximum e(x|y) value.



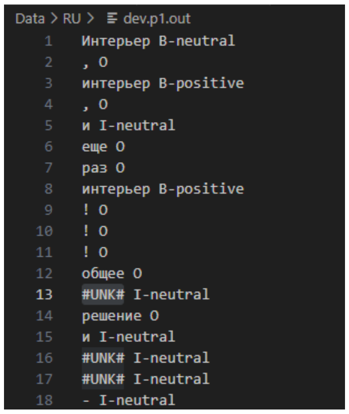
```
Part 1 Question 3 Answers - ES Data
          x  count_y_to_x    e(x|y)  count y           y  count_y_plus_k
0         !         158.0  0.005442  29035.0           O         29036.0
1         "           2.0  0.011628    171.0  I-negative           172.0
2     #UNK#           1.0  0.028571     34.0   I-neutral            35.0
3         %          12.0  0.000413  29035.0           O         29036.0
4         (         131.0  0.004512  29035.0           O         29036.0
..      ...           ...       ...      ...         ...             ...
945       ’           8.0  0.000276  29035.0           O         29036.0
946       “           1.0  0.003175    314.0  I-positive           315.0
947       ”           1.0  0.003175    314.0  I-positive           315.0
948       …           1.0  0.002618    381.0  B-negative           382.0
949       €           1.0  0.003175    314.0  I-positive           315.0

[950 rows x 6 columns]
Part 1 Question 3 Answers - RU Data
            x  count_y_to_x    e(x|y)  count y          y  count_y_plus_k
0           !         888.0  0.021911  40526.0          O         40527.0
1           "           4.0  0.057971     68.0  I-neutral            69.0
2       #UNK#           1.0  0.020408     48.0  I-neutral            49.0
3           %          12.0  0.000296  40526.0          O         40527.0
4           (         271.0  0.006687  40526.0          O         40527.0
...       ...           ...       ...      ...        ...             ...
1593     язык           1.0  0.000025  40526.0          O         40527.0
1594    яйцом           1.0  0.000025  40526.0          O         40527.0
1595   января           3.0  0.000074  40526.0          O         40527.0
1596        –          10.0  0.000247  40526.0          O         40527.0
1597        …           5.0  0.000123  40526.0          O         40527.0
```



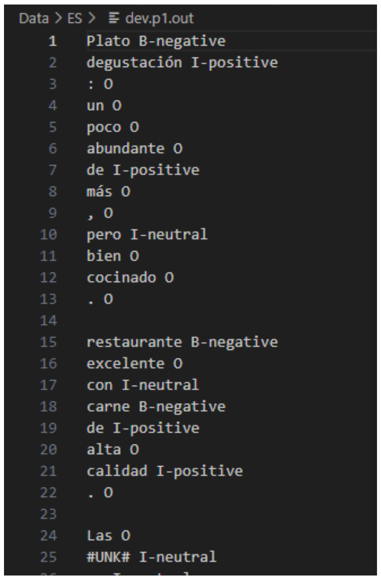
```python
def generate_y_values_with_dev_in_y_star(file_path_dev_in,file_path_train,file_path_dev_p1_out):
    df_dev_in_y_star=create_df_x_to_y_star_with_train_and_dev_in(file_path_dev_in,file_path_train)
    #display(df_dev_in_y_star)
    df_train=file_to_df(file_path_train)
    x_values=df_train['x'].tolist()
    #x_values_df_dev_in=df_dev_in_y_star['x'].tolist()
    #print("Plato" in x_values_df_dev_in)
    with open(file_path_dev_in, 'r',encoding='utf-8') as file:
        lines = file.readlines()
        for l in range(len(lines)):
            line=lines[l].strip()
            if line in x_values:
                # print(line)
                possible_y_values=df_dev_in_y_star[df_dev_in_y_star['x'] == line]['y'].tolist()
                lines[l]=line+" "+possible_y_values[0]

                if (len(possible_y_values)!=1):
                    print ("something wrong: x_vlaues in df_dev_in_y_star not unique for some reason,for line: ",l,line,possible_y_values)
            else:
                if (line!=""):
                    #  print("here")
                    x_value="#UNK#"
                    y_values=df_dev_in_y_star[df_dev_in_y_star['x'] == x_value]['y'].tolist()
                    line="#UNK#"+" "+y_values[0]
                    lines[l]=line
                else:
                    lines[l]=""

    with open(file_path_dev_p1_out, 'w',encoding='utf-8') as file:

        for line in lines:
            file.write(line+"\n")
```

In this function we just create the df_dev_in_y_star, basically applying the train dataframe model parameters onto dev_in dataframe. And then we read the dev.in file, and for every line we look up its corresponding y value in df_dev_in_y_star and adding it to the line. Values that don’t appear in train dataset will be replaced with #UNK# and the y value in df_dev_in_y_star for x value=#UNK# will be applied to it. Then, we will write the lines into file_path_dev_p1_out.

Here are what dev.p1.out files looks like:

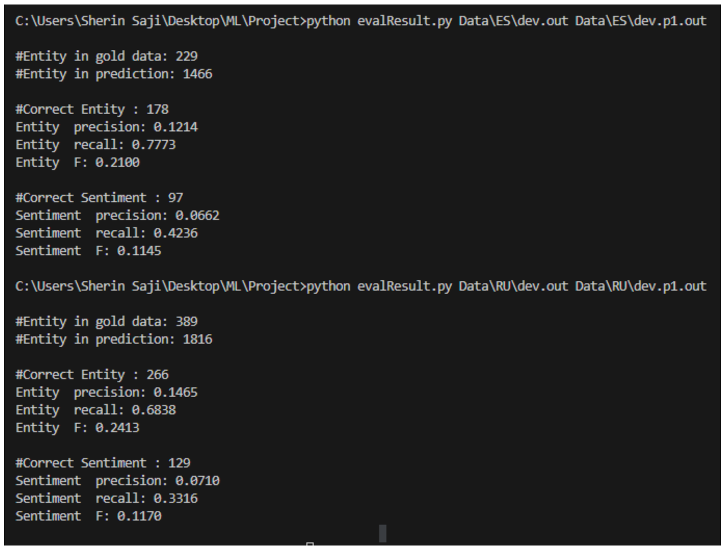


```
Интерьер B-neutral
, O
интерьер B-positive
, O
и I-neutral
еще O
раз O
интерьер B-positive
! O
! O
! O
общее O
#UNK# I-neutral
решение O
и I-neutral
#UNK# I-neutral
#UNK# I-neutral
- I-neutral
```



```
Plato B-negative
degustación I-positive
: O
un O
poco O
abundante O
de I-positive
más O
, O
pero I-neutral
bien O
cocinado O
. O

restaurante B-negative
excelente O
con I-neutral
carne B-negative
de I-positive
alta O
calidad I-positive
. O

Las O
#UNK# I-neutral
```

Here is comparison against the goal standard dev.out files:



```
C:\Users\Sherin Saji\Desktop\ML\Project>python evalResult.py Data\ES\dev.out Data\ES\dev.p1.out

#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

C:\Users\Sherin Saji\Desktop\ML\Project>python evalResult.py Data\RU\dev.out Data\RU\dev.p1.out

#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

Transition Probabilities Calculation

In this section, the compute_probabilities_v2 function computes the essential probabilities for the HMM, namely initial transition probabilities and transition probabilities. The function takes as input the training data and a list of possible states. It initializes dictionaries to store counts for various events, such as starting transitions, transitions from one state to another. The code then iterates through each sentence in the provided data, extracting words and corresponding states (tags) using regular expressions. It incrementally updates the counts for different events based on the observed transitions and emissions. Finally, it calculates the probabilities by dividing these counts by relevant totals.

```python
def compute_probabilities_v2(data, state_list):

    # Create a dictionary to store the count of starting transitions, count of transitions from one state to other
    # coutn of emissions for each state and count of occurences for each state and count of occurences for each state
    start_transition_count = {state: 0 for state in state_list}
    transition_count = {state: {state2: 0 for state2 in state_list} for state in state_list}
    emission_count = {state: {} for state in state_list}
    state_count = {state: 0 for state in state_list}

    # Iterate over each sentence in the provided data
    for sentence in data:
        # Initialize a variable to keep track of the previous state in the sentence
        prev_state = None
        # Iterate over each line (word and tag pair) in the sentence
        for line in sentence:
            # Use regular expression to extract word and state from the line
            match = re.search(r'^(.*)\s(\S+)$', line.strip())
            # Check if the regular expression match was successful
            if match:
                # Extract word and state using groups() method of the match object
                word, state = match.groups()
                # Check if this is the beginning of a sentence (no previous state)
                if prev_state is None:
                    # Increment the count of starting transitions for the current state
                    start_transition_count[state] += 1
                else:
                    # Increment the count of transitions from the previous state to the current state
                    transition_count[prev_state][state] += 1
                    # Increment the emission count for the previous state and the current word
                    # If the word has not been encountered before for the current state, initialize the count to 0
                    emission_count[prev_state][word] = emission_count[prev_state].get(word, 0) + 1
                # Increment the count of occurrences for the current state
                state_count[state] += 1
                # Update the previous state for the next iteration
                prev_state = state
        # After processing all lines in the sentence, check if there is a previous state
        if prev_state:
            # Increment the emission count for the last state and word
            # If the word has not been encountered before for the last state, initialize the count to 0
            emission_count[prev_state][word] = emission_count[prev_state].get(word, 0) + 1

    # Calculate the total number of sentences in the data
    total_sentences = len(data)
    # Calculate the probabilities for start transitions based on the counts
    start_transition_prob = {state: count / total_sentences for state, count in start_transition_count.items()}
    # Calculate the probabilities for transitions based on the counts
    transition_prob = {state: {state2: count2 / state_count[state] for state2, count2 in count.items()} for state, count in
    # Calculate the probabilities for emissions (words) based on the counts
    emission_prob = {state: {word: count / state_count[state] for word, count in state_emission_count.items()} for state, st
    # Return the calculated probabilities for start transitions, transitions, and emissions
    return start_transition_prob, transition_prob, emission_prob
```

Viterbi Algorithm

DP section

Firstly, we initialize a table called “df_incoming” to store **backwards pointers** to the most likely preceding state and a series called “step_probabilities” to store the overall probabilities of reaching each state in the latest step.

For each current state, a nested loop iterates through the previous states. In this loop, the algorithm calculates the probability of reaching the current state from each previous state using the formula $\pi(k, v) = \max_{u \in \mathcal{T}} \{\pi(k-1, u) \cdot a_{u,v} \cdot b_v(x_k)\}$. This involves multiplying the previous step probability, the transition probability from the previous state to the current state, and the emission probability of the current observation given the current state.

```python
def viterbi(observation, transition, emission, start_probabilities):

    # Extract states from the transition DataFrame
    states = transition.index.tolist()
    state_count = transition.shape[0]
    step_count = len(observation)

    # Create a DataFrame to memoize the incoming edges for each node in the Viterbi trellis diagram
    df_incoming = np.zeros((state_count , step_count), dtype=int)
    df_incoming = pd.DataFrame(df_incoming, index = states)

    # Create a series to memoize the overall probabilities of arriving at each state at a particular step
    # Calculate the step probabilities using emission and start probabilities
    try:
        step_probabilities = emission[observation[0]] * start_probabilities
    except:
        step_probabilities = emission["#UNK#"] * start_probabilities
    step_probabilities_temp = pd.Series({state: 0 for state in states})

    # Iterate over each step in the observation sequence (excluding the first step)
    for step in range(1, step_count):
        # Iterate over each current state
        for current_state in states:
            # Create a Series to track probabilities of choosing the maximum path from previous states to the current state
            choosing_max = pd.Series({state: 0 for state in states})
            # Iterate over each previous state
            for previous_state in states:
                try:
                    # Calculate the probability of choosing the maximum path using Viterbi algorithm formula
                    # Multiply step probability, transition probability, and emission probability
                    choosing_max.loc[previous_state] = step_probabilities.loc[previous_state] * transition.loc[previous_sta
                except:
                    # If emission probability for the current observation is not available, use emission for "#UNK#"
                    choosing_max.loc[previous_state] = step_probabilities.loc[previous_state] * transition.loc[previous_sta
            # Find the state with the maximum calculated probability
            max_state = choosing_max.idxmax()
            # Update the step_probabilities_temp Series with the maximum probability for the current state
            step_probabilities_temp.loc[current_state] = choosing_max.loc[max_state]
            # Memoize the incoming state that leads to the maximum probability in the df_incoming DataFrame
            df_incoming.loc[current_state, step] = max_state
        # Update the step_probabilities Series for the next step using the updated probabilities from step_probabilities_te
        step_probabilities = step_probabilities_temp.copy()
```

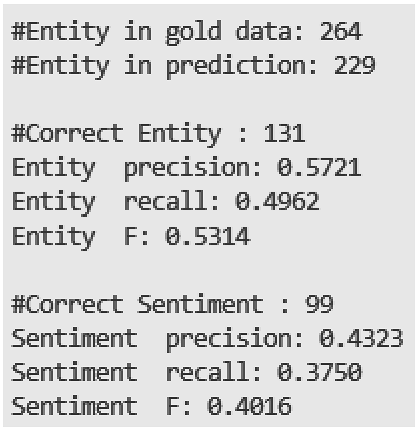
Backtracking

After probabilities have been calculated for all steps, the algorithm identifies the most likely ending state and backtracks through the backwards pointers (df_incoming) to determine the sequence of states that maximizes the overall likelihood of the observation sequence.

```python
# get sequences in descending order of likelihood
descending_final_probability_states = step_probabilities.sort_values(ascending=False).index.to_numpy()
# Create a DataFrame to store sequences of states
df_sequences = np.zeros((state_count , step_count), dtype=int)
df_sequences = pd.DataFrame(df_sequences)
# Initialize the last step of each sequence with the most likely states
for sequence_rank in range(state_count):
    df_sequences.loc[sequence_rank, step_count-1] = descending_final_probability_states[sequence_rank]
# Iterate through steps and fill in the rest of the sequences using memoized incoming edges
for sequence_rank in range(state_count):
    for step in range(step_count-2, -1, -1):
        # Get the previous state of the current step from the memoized incoming edges
        previous_state = df_sequences.loc[sequence_rank, step+1]
        # Assign the previous state to the current step in the sequence
        df_sequences.loc[sequence_rank, step] = df_incoming.loc[previous_state, step+1]
# Return the sequence of states with the highest likelihood (most likely sequence)
return df_sequences.loc[0].values
```

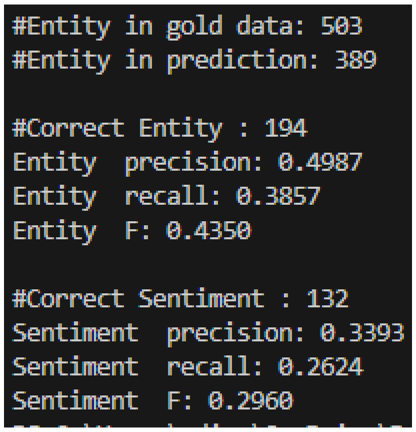
Results

ES/dev.p2.out



```
#Entity in gold data: 264
#Entity in prediction: 229

#Correct Entity : 131
Entity  precision: 0.5721
Entity  recall: 0.4962
Entity  F: 0.5314

#Correct Sentiment : 99
Sentiment  precision: 0.4323
Sentiment  recall: 0.3750
Sentiment  F: 0.4016
```

RU/dev.p2.out



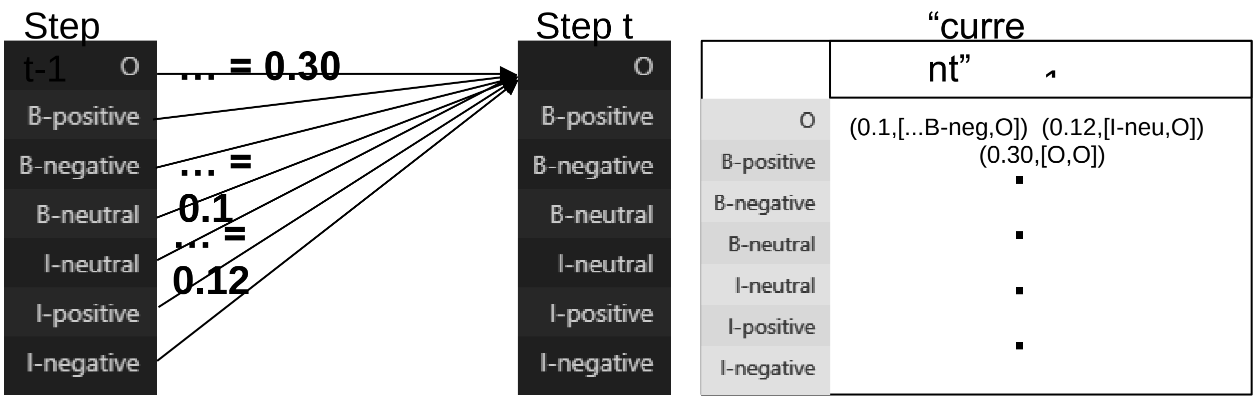
```
#Entity in gold data: 503
#Entity in prediction: 389

#Correct Entity : 194
Entity  precision: 0.4987
Entity  recall: 0.3857
Entity  F: 0.4350

#Correct Sentiment : 132
Sentiment  precision: 0.3393
Sentiment  recall: 0.2624
Sentiment  F: 0.2960
```

Part 3

Extension of Viterbi algorithm

While the normal Viterbi algorithm entails calculating $\pi(k,v) = \max_{u \in \mathcal{T}} \{\pi(k-1,u) \cdot a_{u,v} \cdot b_v(x_k)\}$ and storing the probability and reference to only the most likely preceding node, we can extend it to memoize the k-most likely preceding nodes in a table.

Trellis diagram



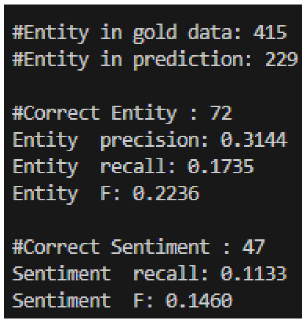
In the algorithm, we keep two tables called “preceding” and “current”.
The rows represent the states and the number of columns correspond with the number of best paths that we want from the algorithm (k).
Each cell contains a tuple, recording the probability of reaching that state and the path taken to get there.
“Preceding” contains the data for the states in the preceding step while “current” contains the data for the states in the current step.
At each step, we reference “preceding” to update the new values for “current”, only keeping the k most probable paths.

The markov property of first order HMMs states that the current state is only dependent on the preceding state, so we only need to reference the data from the previous state and not deeper into history.

By keeping the k-best paths for each state, we ensure that every k-best path is included, even in the case where the k-best paths all converge into a single state.

In the end, we collapse the table into an array and arrange the tuples in ascending order of probability. The kth-best probability and path can be obtained from this array.

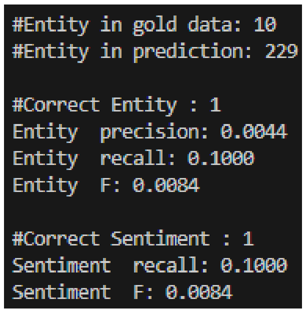
ES/dev.p3.2nd.out



```
#Entity in gold data: 415
#Entity in prediction: 229

#Correct Entity : 72
Entity  precision: 0.3144
Entity  recall: 0.1735
Entity  F: 0.2236

#Correct Sentiment : 47
Sentiment  recall: 0.1133
Sentiment  F: 0.1460
```

As expected, the F-score for the 2nd best path is lower than that of the best path in ES/dev.p2.out

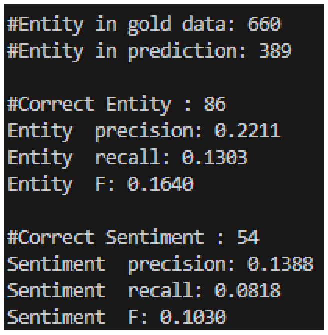
ES/dev.p3.8th.out



```
#Entity in gold data: 10
#Entity in prediction: 229

#Correct Entity : 1
Entity  precision: 0.0044
Entity  recall: 0.1000
Entity  F: 0.0084

#Correct Sentiment : 1
Sentiment  recall: 0.1000
Sentiment  F: 0.0084
```

As expected, the F-score for the 8th best path is lower than that of both the 2nd best path in ES/dev.p3.2nd.out and the best path in ES/dev.p2.out

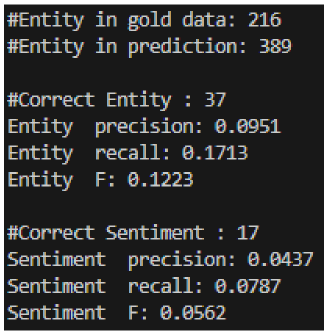
RU/dev.p3.2nd.out



```
#Entity in gold data: 660
#Entity in prediction: 389

#Correct Entity : 86
Entity  precision: 0.2211
Entity  recall: 0.1303
Entity  F: 0.1640

#Correct Sentiment : 54
Sentiment  precision: 0.1388
Sentiment  recall: 0.0818
Sentiment  F: 0.1030
```

As expected, the F-score for the 2nd best path is lower than that of the best path in RU/dev.p2.out

RU/dev.p3.8th.out



```
#Entity in gold data: 216
#Entity in prediction: 389

#Correct Entity : 37
Entity  precision: 0.0951
Entity  recall: 0.1713
Entity  F: 0.1223

#Correct Sentiment : 17
Sentiment  precision: 0.0437
Sentiment  recall: 0.0787
Sentiment  F: 0.0562
```

As expected, the F-score for the 8th best path is lower than that of both the 2nd best path in RU/dev.p3.2nd.out and the best path in RU/dev.p2.out

Part 4

Limitations of first order HMM

Though the first order HMM does work well in predicting the transition and emission state probabilities well to some extent, there are many limitations with respect to the model itself, which are described as follows:

1. **Memoryless property of the first - order HMM :** As shown in the figure 1 below, the the current hidden state is dependent only on the previous hidden state and does not carry any information with regards to the historical hidden states prior to the previous hidden state.

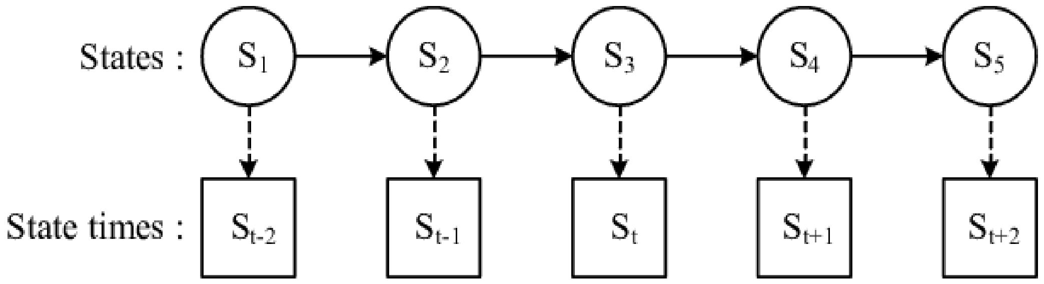


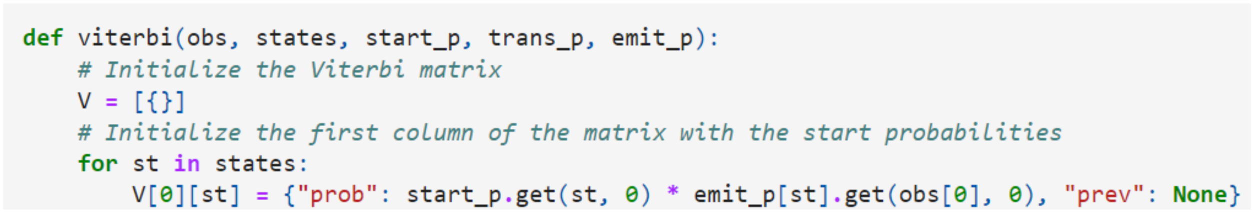
Figure 1: First - Order HMM *(Familua, Ayokunle & Ndjiongue, A & Ogunyanda, Kehinde & Cheng, Ling & Ferreira, Hendrik & Swart, Theo. (2015))*

2. **Inability to express dependencies between hidden states:** As the first order HMM has a lot of unstructured parameters, there is a high probability that these parameters are conditionally dependent on not only the previous hidden parameter but also other historical hidden states, whereby there are higher order correlations amongst the hidden states. One such application is protein - folding (*Seifert M, Abou-El-Ardat K, Friedrich B, Klink B, Deutsch A (2014)*)

Even though the second order HMM might not fully capture the dependencies amongst all the historical states, the model is able to make use of more contextual information compared to the first-order HMM as it makes use of transition probabilities from the two previous states instead of the previous state alone.

Implementation of 2nd-order HMM

The 2nd order HMM differs from the first order HMM such that the transition probabilities depend on the two previous states instead of just a single state. To achieve this, there will be changes in the arguments passed to the Viterbi algorithm as shown in Figure 2 with respect to the transition probabilities (*trans_p* argument in Figure 2).



```python
def viterbi(obs, states, start_p, trans_p, emit_p):
    # Initialize the Viterbi matrix
    V = [{}]
    # Initialize the first column of the matrix with the start probabilities
    for st in states:
        V[0][st] = {"prob": start_p.get(st, 0) * emit_p[st].get(obs[0], 0), "prev": None}
```

Figure 2: Arguments of the Viterbi algorithm (*obs* refers to observations (list); *states* refer to states (list); start_p refers to the start state probabilities; emit_p refers to the emission probabilities and trans_p refers to the transition probabilities.

While in the first-order HMM, we represent the transition probabilities (*trans_p)* as a dictionary whose keys are the previous states and the values being the transition probabilities of these prior states, the second-order HMM differs such that the keys are the first previous state; the values are another nested dictionary, which in turn contains keys that correspond to the second previous state. To compute the transition probabilities, we make use of the *compute_probabilities* function, which has been documented in *Appendix A*.

The *compute_probabilities* function consists of three main parts, which are:

1. Initialization of Counts: The function begins by setting up the structures (start_count, transition_count, emission_count) to hold counts for the initial states, transitions between states, and emissions from states to observations (words).
2. Populating the Counts: As the function processes the training data (sentences), it updates these count structures based on the observed sequences of states and words.
3. Computation of Probabilities: Once all the counts have been gathered, the function computes the corresponding probabilities (chances) by dividing each individual count by the appropriate total.

Compute_probabilites function

Initialization of the start, transition and emission counts

Input Arguments

In this step, the *start_count* is a single state dictionary of zero values for every state in the argument *state_list*. The *state_list* argument is the list of all the hidden states/tags involved:

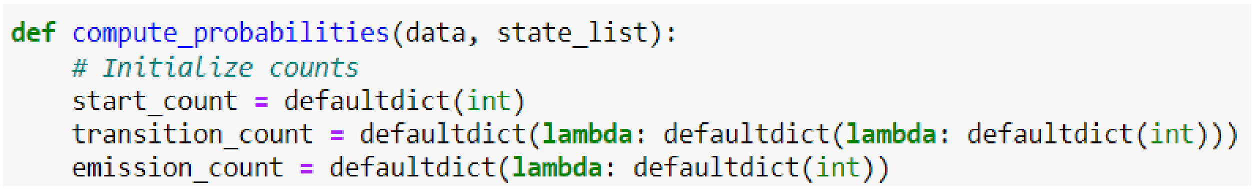
['O', 'B-positive', 'B-negative': 381, 'B-neutral', 'I-neutral', ''I-positive', 'I-negative']

The *data* variable will be a list of lists, whereby the outer lists refer to the sentences delimited by '\n\n' and the inner lists refer to the words within delimited by a '\n' and have been loaded through the *load_data* method in Part 2.

Initial states of the start, emission and transition counts

The *start_count* variable is initialized as a dictionary, whose length is equal to the *state_list* argument and will be used in the code later on to compute the start transition probabilities denoted by *start_p.* The start transition probabilities are indicated as a single dictionary instead of nested dictionaries like the normal transition probabilities because these probabilities represent the initial distribution of states before any observations are made. Once the first observation is made and the model transitions to the next state, the current and the previous state pairs are used as a basis for calculating the transition probabilities for the subsequent conditions.

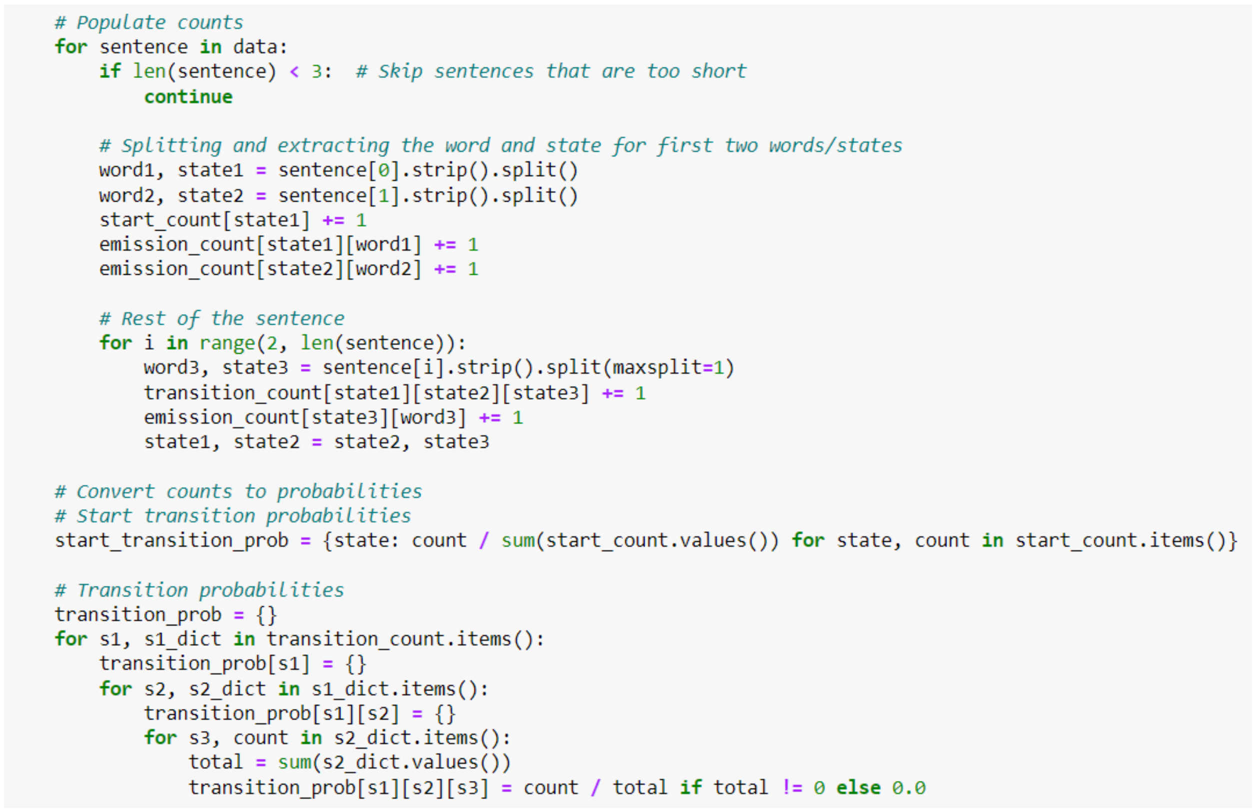
The emission count that will give rise to the emission probabilities, *emission_count, and* initialized are similar to the first-order HMM, with the nested dictionaries representing the observation given the current state. The transition_count for our second-order Hidden Markov Model (HMM) is a three-tiered dictionary capturing state sequences of the form: state1 -> state2 -> state3. It counts how often each state triplet occurs in training data, reflecting dependencies over three consecutive states. This structure enriches the HMM, allowing it to recognize longer patterns in sequences, but it also omits sentences shorter than three words.



```python
def compute_probabilities(data, state_list):
    # Initialize counts
    start_count = defaultdict(int)
    transition_count = defaultdict(lambda: defaultdict(lambda: defaultdict(int)))
    emission_count = defaultdict(lambda: defaultdict(int))
```

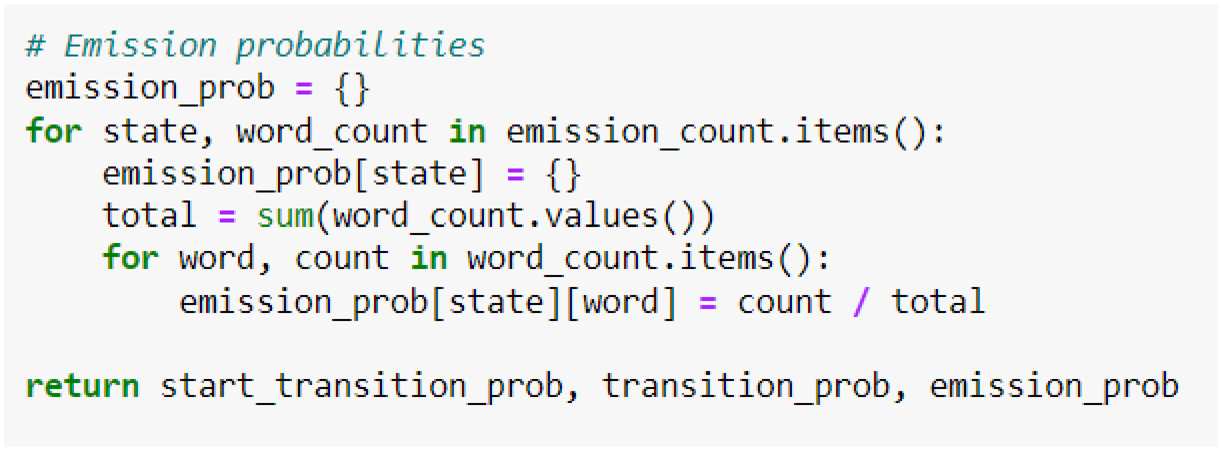
Figure 3: Initialization of the start,transition and emission probabilities

Populating the counts of transition and emission at the current states



```python
# Populate counts
for sentence in data:
    if len(sentence) < 3:  # Skip sentences that are too short
        continue

    # Splitting and extracting the word and state for first two words/states
    word1, state1 = sentence[0].strip().split()
    word2, state2 = sentence[1].strip().split()
    start_count[state1] += 1
    emission_count[state1][word1] += 1
    emission_count[state2][word2] += 1

    # Rest of the sentence
    for i in range(2, len(sentence)):
        word3, state3 = sentence[i].strip().split(maxsplit=1)
        transition_count[state1][state2][state3] += 1
        emission_count[state3][word3] += 1
        state1, state2 = state2, state3

# Convert counts to probabilities
# Start transition probabilities
start_transition_prob = {state: count / sum(start_count.values()) for state, count in start_count.items()}

# Transition probabilities
transition_prob = {}
for s1, s1_dict in transition_count.items():
    transition_prob[s1] = {}
    for s2, s2_dict in s1_dict.items():
        transition_prob[s1][s2] = {}
        for s3, count in s2_dict.items():
            total = sum(s2_dict.values())
            transition_prob[s1][s2][s3] = count / total if total != 0 else 0.0
```

Figure 4: Code blocks to compute the counts of start transition and transition at current states.



```python
# Emission probabilities
emission_prob = {}
for state, word_count in emission_count.items():
    emission_prob[state] = {}
    total = sum(word_count.values())
    for word, count in word_count.items():
        emission_prob[state][word] = count / total

return start_transition_prob, transition_prob, emission_prob
```

Figure 5: Code blocks to compute the counts of emission at current states.

The explanation for populating counts of transition and emission at current state is structured according to the comments shown on figure 4 and 5:

1)The if clause that skips words that are too short.
In this code block given in *Figure 4*, the function iterates over each sentence in the provided data. If the sentence has fewer than 3 words, it's skipped because for a second-order HMM, we need at least three states to calculate the transition probability.

2)Splitting and extracting the word and state for the first two words
Additionally,figure4 also shows that the function extracts the first two words and their corresponding states from the sentence. The start_count for the first state is incremented, indicating the number of times the sequence starts with that particular state. Additionally, the emission counts for these two states are updated based on the words they emit.

3)Rest of the sentence
For the remaining parts of the sentence (from the third word/state onward), the function updates the transition and emission counts:

- Transition Counts: The code updates the counts of observing a transition from state1 to state2 and then to state3. This is essential for a second-order HMM, as the transition probability is based on the two preceding states.
- Emission Counts: The emission count for state3 emitting word3 is incremented.
- State Update: Finally, to slide the window for the next iteration, state1 is updated to the previous state2 and state2 to the current state3.

4)Convert counts to probabilities/Start transition probabilities
This line computes the probability of a sequence starting with a particular state. It divides the count of each state (how many sequences start with that state) by the total number of sequences. The result is stored in a dictionary named start_transition_prob.

5)Transition probabilities
This section is responsible for converting the counts of state transitions into probabilities:

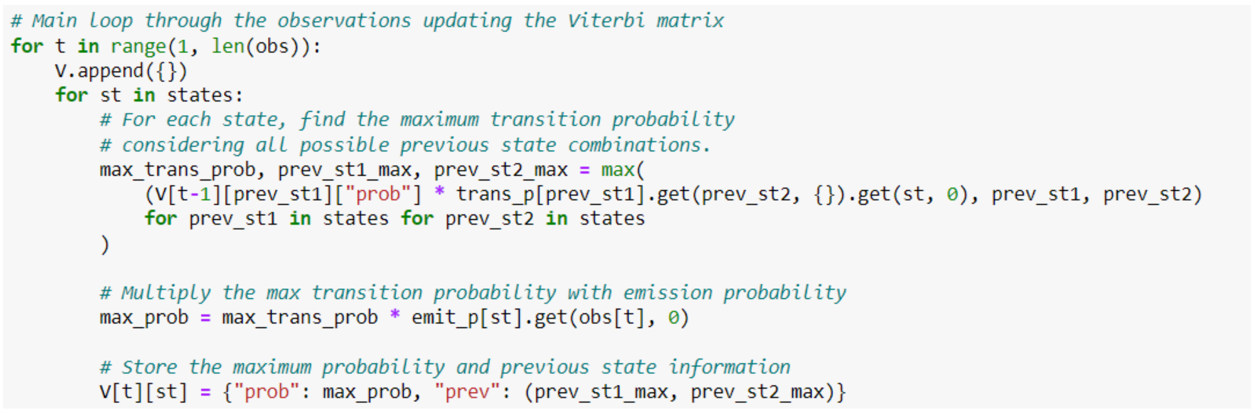
- The outer loop iterates through each initial state (s1).
- The next inner loop moves through the subsequent state (s2) that follows s1.
- The innermost loop then considers the state (s3) that comes after the pair (s1, s2). For each of these s3 states, the code divides its count by the total counts of all states that follow the pair (s1, s2). If the total count is zero (which should be rare), it ensures the result is 0.0 to avoid a division by zero.

6)Emission probabilities
This segment calculates the emission probabilities:

- It iterates over each state and the words that are emitted by that state (along with their counts).
- For each state, it calculates the probability of emitting a particular word by dividing the count of that word by the total counts of all words emitted by the state.

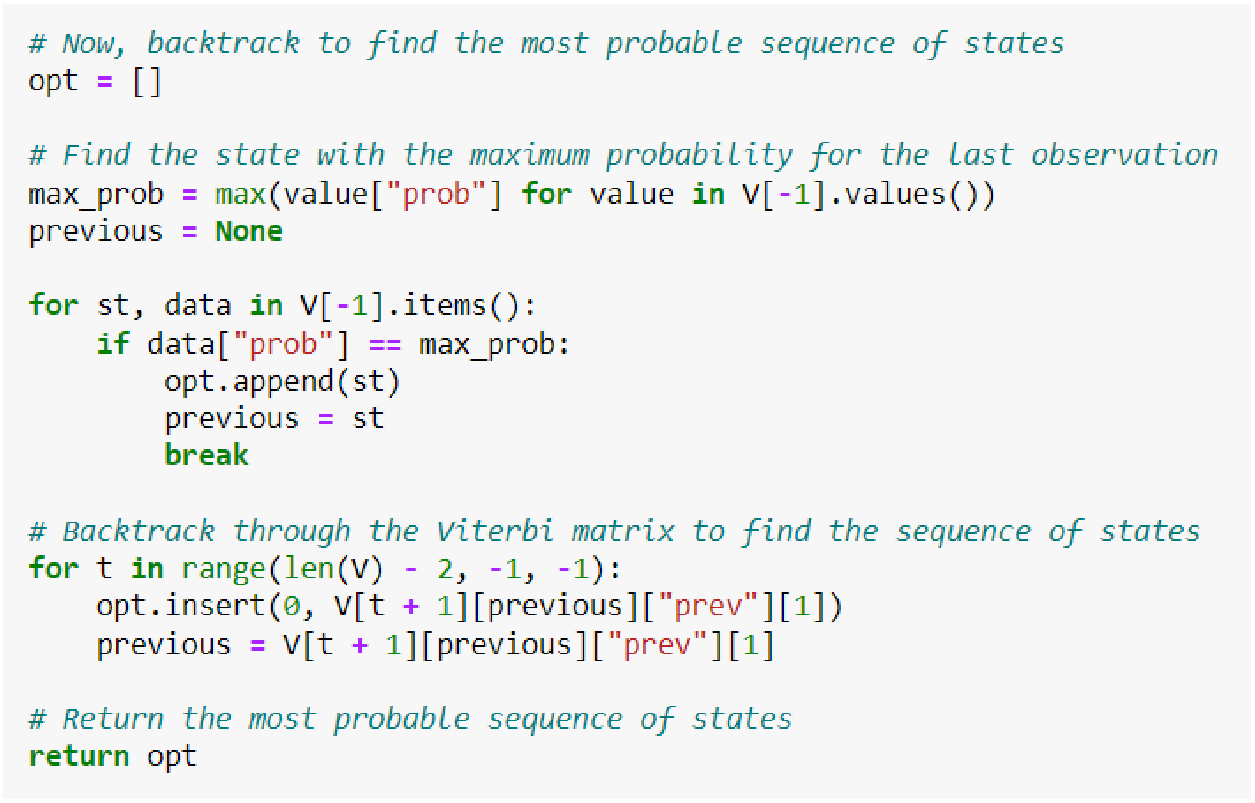
Finally, the function returns the three types of probabilities (start, transition, and emission) as separate dictionaries.

In essence, this code segment is about converting raw counts into probabilities by normalizing the counts with the respective totals. These probabilities will be crucial when the HMM model is used to predict sequences.



```python
# Main loop through the observations updating the Viterbi matrix
for t in range(1, len(obs)):
    V.append({})
    for st in states:
        # For each state, find the maximum transition probability
        # considering all possible previous state combinations.
        max_trans_prob, prev_st1_max, prev_st2_max = max(
            (V[t-1][prev_st1]["prob"] * trans_p[prev_st1].get(prev_st2, {}).get(st, 0), prev_st1, prev_st2)
            for prev_st1 in states for prev_st2 in states
        )

        # Multiply the max transition probability with emission probability
        max_prob = max_trans_prob * emit_p[st].get(obs[t], 0)

        # Store the maximum probability and previous state information
        V[t][st] = {"prob": max_prob, "prev": (prev_st1_max, prev_st2_max)}
```

Figure 6: Code blocks to   explain how max_prob is derived.



```python
# Now, backtrack to find the most probable sequence of states
opt = []

# Find the state with the maximum probability for the last observation
max_prob = max(value["prob"] for value in V[-1].values())
previous = None

for st, data in V[-1].items():
    if data["prob"] == max_prob:
        opt.append(st)
        previous = st
        break

# Backtrack through the Viterbi matrix to find the sequence of states
for t in range(len(V) - 2, -1, -1):
    opt.insert(0, V[t + 1][previous]["prev"][1])
    previous = V[t + 1][previous]["prev"][1]

# Return the most probable sequence of states
return opt
```

Figure 7: Code blocks to whow viterbi optimization

The explanation for the viterbi algorithm is structured according to the comments shown on figure 2 and 6:

1)Initialize Viterbi Matrix and the first column of the matrix with the start probabilities
The algorithm starts by initializing the first set of probabilities, V[0], for each state. These probabilities are based on the starting probabilities start_p and the emission probabilities emit_p for the first observation obs[0].

2)Main loop through the observations updating the viterbi matrix
We now enter the main part of the Viterbi algorithm, looping over each observation in the input sequence (from the second observation onward since the first is already handled).

3)For each state find the maximum transition probability considering all possible previous state combinations
This part calculates the maximum transition probability to reach a state from all possible combinations of the two previous states. It does this for every state and at every time step.

4)Multiply the max transition probability with the emission probability.
After finding the maximum transition probability, we need to account for the probability of the state emitting the current observation. This is done by multiplying the max_trans_prob with the respective emission probability from the current state.

5)Store the maximum probability and previous state information
For each state and at each time step, this segment stores two pieces of information: the calculated maximum probability max_prob and a tuple indicating which two states from the previous time step contributed to this maximum probability.

6)Backtrack to find the most probable sequence of states
With the Viterbi matrix filled out, we now backtrack to determine the most probable sequence of states that could have resulted in the observed sequence.

7)Find the state with the maximum probability for the last observation
We start backtracking from the last observation. To do this, first, we need to determine which state at the final observation has the highest probability.

8)Backtrack through the viterbi matrix to find the sequence of states and return the most probable sequence of states
This loop backtracks through the Viterbi matrix, from the penultimate observation to the first, reconstructing the most probable sequence of states.
By understanding each of these segments, you get a comprehensive view of how the Viterbi algorithm determines the most probable state sequence for a given observation sequence.

APPENDICES

PART IV: COMPUTE_PROBABILITIES FUNCTION

```
from collections import defaultdict

def compute_probabilities(data, state_list):
    # Initialize counts
    start_count = defaultdict(int)
    transition_count = defaultdict(lambda: defaultdict(lambda: defaultdict(int)))
    emission_count = defaultdict(lambda: defaultdict(int))

    # Populate counts
    for sentence in data:
        if len(sentence) < 3:  # Skip sentences that are too short
```

```
            continue

        # Splitting and extracting the word and state for first two words/states
        word1, state1 = sentence[0].strip().split()
        word2, state2 = sentence[1].strip().split()
        start_count[state1] += 1
        emission_count[state1][word1] += 1
        emission_count[state2][word2] += 1

        # Rest of the sentence
        for i in range(2, len(sentence)):
            word3, state3 = sentence[i].strip().split(maxsplit=1)
            transition_count[state1][state2][state3] += 1
            emission_count[state3][word3] += 1
            state1, state2 = state2, state3

    # Convert counts to probabilities
    # Start transition probabilities
    start_transition_prob = {state: count / sum(start_count.values()) for state, count in start_count.items()}

    # Transition probabilities
    transition_prob = {}
    for s1, s1_dict in transition_count.items():
        transition_prob[s1] = {}
        for s2, s2_dict in s1_dict.items():
            transition_prob[s1][s2] = {}
            for s3, count in s2_dict.items():
                total = sum(s2_dict.values())
                transition_prob[s1][s2][s3] = count / total if total != 0 else 0.0

    # Emission probabilities
    emission_prob = {}
    for state, word_count in emission_count.items():
        emission_prob[state] = {}
        total = sum(word_count.values())
        for word, count in word_count.items():
            emission_prob[state][word] = count / total

    return start_transition_prob, transition_prob, emission_prob
```

PART IV: VITERBI ALGORITHM

```
def viterbi(obs, states, start_p, trans_p, emit_p):
    # Initialize the Viterbi matrix.
    V = [{}]

    # Initialize the first column of the matrix with the start probabilities
    for st in states:
        V[0][st] = {"prob": start_p.get(st, 0) * emit_p[st].get(obs[0], 0), "prev": None}

    # Main loop through the observations updating the Viterbi matrix
    for t in range(1, len(obs)):
        V.append({})
        for st in states:
            # For each state, find the maximum transition probability
            # considering all possible previous state combinations.
            max_trans_prob, prev_st1_max, prev_st2_max = max(
                (V[t-1][prev_st1]["prob"] * trans_p[prev_st1].get(prev_st2, {}).get(st, 0), prev_st1, prev_st2)
            for prev_st1 in states for prev_st2 in states
        )

        # Multiply the max transition probability with emission probability
        max_prob = max_trans_prob * emit_p[st].get(obs[t], 0)

        # Store the maximum probability and previous state information
        V[t][st] = {"prob": max_prob, "prev": (prev_st1_max, prev_st2_max)}

    # Now, backtrack to find the most probable sequence of states
    opt = []

    # Find the state with the maximum probability for the last observation
    max_prob = max(value["prob"] for value in V[-1].values())
    previous = None

    for st, data in V[-1].items():
        if data["prob"] == max_prob:
            opt.append(st)
            previous = st
            break

    # Backtrack through the Viterbi matrix to find the sequence of states
    for t in range(len(V) - 2, -1, -1):
        opt.insert(0, V[t + 1][previous]["prev"][1])
        previous = V[t + 1][previous]["prev"][1]

    # Return the most probable sequence of states
    return opt
```

For example, in part-of-speech tagging, using second-order approximations for both contextual and lexical probabilities can increase the accuracy of the tagger to state-of-the-art levels3. These approximations make use of more contextual information than standard statistical systems and new methods of smoothing the estimated probabilities are also introduced to address the sparse data problem3.

Overall, second-order HMMs provide a way to incorporate more information into the model and can improve its performance in certain applications. However, they also introduce additional complexity and computational cost, so their use should be carefully considered depending on the specific problem at hand. Is there anything else you would like to know?

One limitation of first-order HMMs is that they assume the probability of transitioning from one state to another depends only on the current state. This is known as the Markov property or memoryless property1. However, this assumption can be limiting in some applications where the probability of transitioning to a new state depends on more than just the current state.

Second-order HMMs address this limitation by allowing the transition probability of a state at any time to depend on the two previous states at the time2. This allows the model to capture more complex dependencies between states and can improve its performance in certain applications.

In addition to the memoryless property, first-order HMMs are also limited by their inability to express dependencies between hidden states1. This can be a problem in applications where there are higher-order correlations among the hidden states. For example, in protein folding, proteins fold into complex 3-D shapes that determine their function. A first-order HMM may not be able to capture these higher-order correlations among amino acids1.

Overall, second-order HMMs provide a way to incorporate more information into the model and can improve its performance in certain applications. However, they also introduce additional complexity and computational cost, so their use should be carefully considered depending on the specific problem at hand. Is there anything else you would like to know? 😊

REFERENCES

1. Familua, Ayokunle & Ndjiongue, A & Ogunyanda, Kehinde & Cheng, Ling & Ferreira, Hendrik & Swart, Theo. (2015). A Semi-Hidden Markov Modeling of a Low Complexity FSK-OOK In-House PLC and VLC Integration.
2. Article Source: **Autoregressive Higher-Order Hidden Markov Models: Exploiting Local Chromosomal Dependencies in the Analysis of Tumor Expression Profiles** Seifert M, Abou-El-Ardat K, Friedrich B, Klink B, Deutsch A (2014) Autoregressive Higher-Order Hidden Markov Models: Exploiting Local Chromosomal Dependencies in the Analysis of Tumor Expression Profiles. PLOS ONE 9(6): e100295. https://doi.org/10.1371/journal.pone.0100295