Machine Learning Project - Group 40

Phang Teng Fone 1003296
Nicole Lee 1003591
Gerald Lim 1003371

Part 1:

Annotation of dataset individually

Part 2:

2.1

```python
def emission(train_data):
    # Total Number of Labels (DF)
    df_num_labels = train_data['Labels'].value_counts().rename_axis('Labels').reset_index(name='CountY')

    # Total Number of y -> x
    df_emission = train_data.groupby(["Text","Labels"]).size().reset_index()
    df_emission.columns = ["Text", "Labels", "CountY->X"]

    # e(x|y)
    emission_output = pd.merge(df_emission,df_num_labels, on = "Labels")
    emission_output["Emission"] = emission_output["CountY->X"]/emission_output["CountY"]

    return emission_output
```

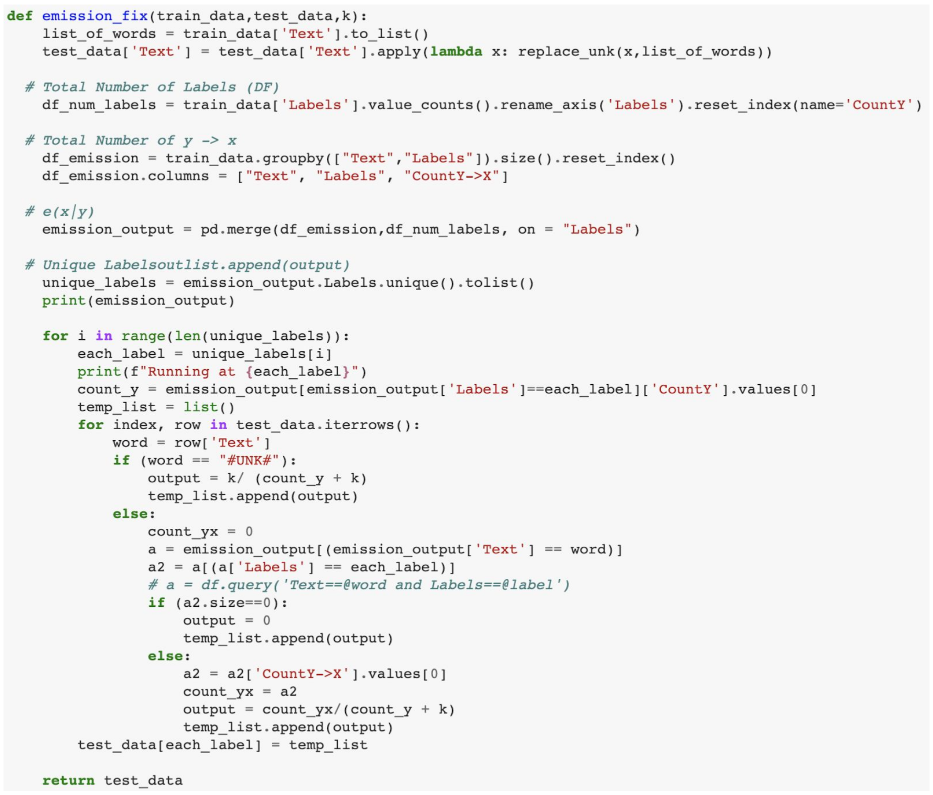
Before running the function emission, we preprocessed the original dataset into a pandas dataframe. We noticed that there was a slight error in one of the dataset where the input of a line has 3 values instead of 2 as such, we have assumed that the first 2 values are meant to be a word thus the code was included to be i[0] + i[1]. However after a few days, the TA clarified that it was a typo, after recalculating with the new dataset, it is still the same as such we have left the code there. This emission function calculates the emission parameters from the training set using MLE:

$$e(x|y) = \frac{\text{Count}(y \to x)}{\text{Count}(y)}$$

```
                           Text      Labels  CountY->X  CountY
0                             !   B-neutral          1   24868
1                             "   B-neutral          4   24868
2                      #04-213   B-neutral          1   24868
3      #2017TaipeiUniversiade   B-neutral          1   24868
4                          #3D   B-neutral          1   24868
...                         ...         ...        ...     ...
50429                    youuu  B-negative          1    1139
50430                 zenyatta  B-negative          2    1139
50431                    zhong  B-negative          1    1139
50432                     zouk  B-negative          1    1139
50433                    zubat  B-negative          1    1139
```

The output of the training data frame is as shown. Where each unique text has its own Count (Y->X) and Count(y) numerals which was derived from the test training data frame.

2.2



```python
def emission_fix(train_data,test_data,k):
    list_of_words = train_data['Text'].to_list()
    test_data['Text'] = test_data['Text'].apply(lambda x: replace_unk(x,list_of_words))

  # Total Number of Labels (DF)
    df_num_labels = train_data['Labels'].value_counts().rename_axis('Labels').reset_index(name='CountY')

  # Total Number of y -> x
    df_emission = train_data.groupby(["Text","Labels"]).size().reset_index()
    df_emission.columns = ["Text", "Labels", "CountY->X"]

  # e(x|y)
    emission_output = pd.merge(df_emission,df_num_labels, on = "Labels")

  # Unique Labelsoutlist.append(output)
    unique_labels = emission_output.Labels.unique().tolist()
    print(emission_output)

    for i in range(len(unique_labels)):
        each_label = unique_labels[i]
        print(f"Running at {each_label}")
        count_y = emission_output[emission_output['Labels']==each_label]['CountY'].values[0]
        temp_list = list()
        for index, row in test_data.iterrows():
            word = row['Text']
            if (word == "#UNK#"):
                output = k/ (count_y + k)
                temp_list.append(output)
            else:
                count_yx = 0
                a = emission_output[(emission_output['Text'] == word)]
                a2 = a[(a['Labels'] == each_label)]
                # a = df.query('Text==@word and Labels==@label')
                if (a2.size==0):
                    output = 0
                    temp_list.append(output)
                else:
                    a2 = a2['CountY->X'].values[0]
                    count_yx = a2
                    output = count_yx/(count_y + k)
                    temp_list.append(output)
        test_data[each_label] = temp_list

    return test_data
```

The emission fix function takes in both the dataframe of both train_data and test_data, we then compared the test dataset with the train dataset, if the test word does not appear in the training set, we introduced a token "#UNK#" (replace_unk function). We then iterate through the training data frame and calculate the new emission probability.

$$e(x|y) = \begin{cases} \frac{\texttt{Count}(y \to x)}{\texttt{Count}(y)+k} & \text{If the word token } x \text{ appears in the training set} \\ \frac{k}{\texttt{Count}(y)+k} & \text{If word token } x \text{ is the special token } \texttt{\#UNK\#} \end{cases}$$

However we noticed that the time taken to iterate through each unique label, then the entire test data frame while comparing for the corresponding count(y) in the second data frame takes O(n^3) which on average takes about 10 minutes to generate the output.

The new emitted data frame where each text will have a probability of different labels

	Text	Paragraph	I-neutral	O	B-negative	B-neutral	B-positive	I-positive	I-negative
0	十年	0	0.000000	0.000114	0.000000	0.000000	0.000000	0.000000	0.00000
1	前	0	0.000285	0.000318	0.000000	0.000000	0.000000	0.000723	0.00000
2	马云	0	0.000000	0.000000	0.000000	0.000618	0.001624	0.000000	0.00000
3	告诉	0	0.000000	0.000216	0.000000	0.000000	0.000000	0.000000	0.00000
4	你	0	0.002851	0.009045	0.018083	0.006484	0.004060	0.001447	0.00738
...	...	...	...	...	...	...	...	...	...
13065	、	343	0.001996	0.007267	0.000000	0.000000	0.000000	0.000000	0.00000
13066	#UNK#	343	0.000143	0.000006	0.001808	0.000154	0.000406	0.000362	0.00369
13067	180	343	0.000000	0.000038	0.000000	0.000000	0.000000	0.000000	0.00000
13068	公里	343	0.000285	0.000267	0.000000	0.000000	0.000000	0.000000	0.00000
13069	。	343	0.000000	0.023642	0.000000	0.000000	0.000000	0.000000	0.00000

2.3

```
df2 = emitted_fix.drop(['Text', 'Paragraph'], axis=1)
df2['max_value'] = df2.max(axis=1)
df2['max_label'] = df2.idxmax(axis=1)
df3 = emitted_fix[['Text']].copy()
df3['Paragraph'] = emitted_fix[['Paragraph']].copy()
df3['max_label'] = df2[['max_label']].copy()
df3 = df3[['Text','max_label','Paragraph']]
```

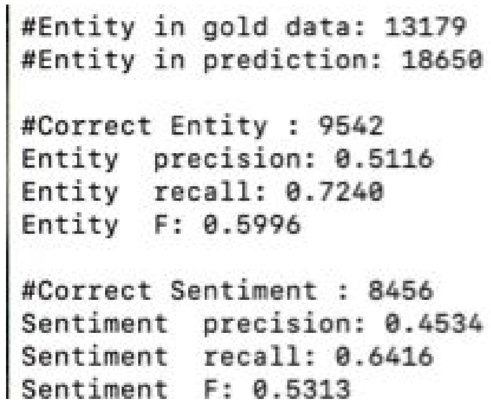
To find the argmax for y*, we used the function in the data frame idxmax to determine the maximum word probability for each word in the sequence then generating a new data frame as shown below which each label and the corresponding paragraph.

```
             Text   max_label  Paragraph
0      Everything           O          0
1          sounds           O          0
2          better           O          0
3            with           O          0
4             the           O          0
...           ...         ...        ...
34186           @           O       2649
34187      Butter  B-positive       2649
34188          My           O       2649
34189       #UNK#  I-negative       2649
34190       #UNK#  I-negative       2649

[34191 rows x 3 columns]
```

Part 2 Results

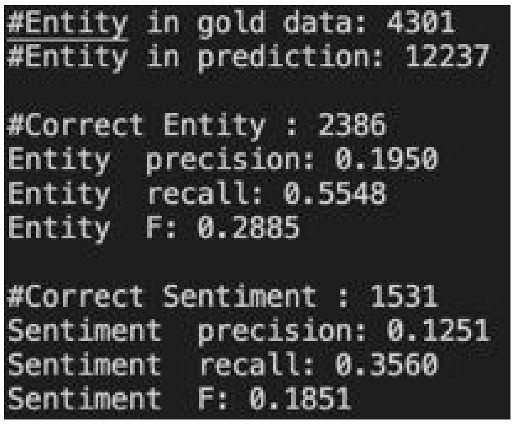
After post processing, we evaluated the score and it is as follows:

EN



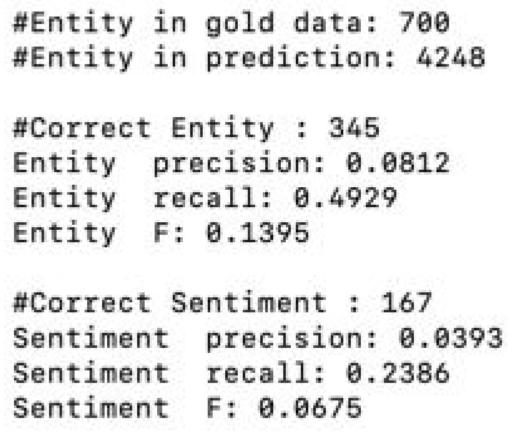
```
#Entity in gold data: 13179
#Entity in prediction: 18650

#Correct Entity : 9542
Entity  precision: 0.5116
Entity  recall: 0.7240
Entity  F: 0.5996

#Correct Sentiment : 8456
Sentiment  precision: 0.4534
Sentiment  recall: 0.6416
Sentiment  F: 0.5313
```

SG



```
#Entity in gold data: 4301
#Entity in prediction: 12237

#Correct Entity : 2386
Entity  precision: 0.1950
Entity  recall: 0.5548
Entity  F: 0.2885

#Correct Sentiment : 1531
Sentiment  precision: 0.1251
Sentiment  recall: 0.3560
Sentiment  F: 0.1851
```

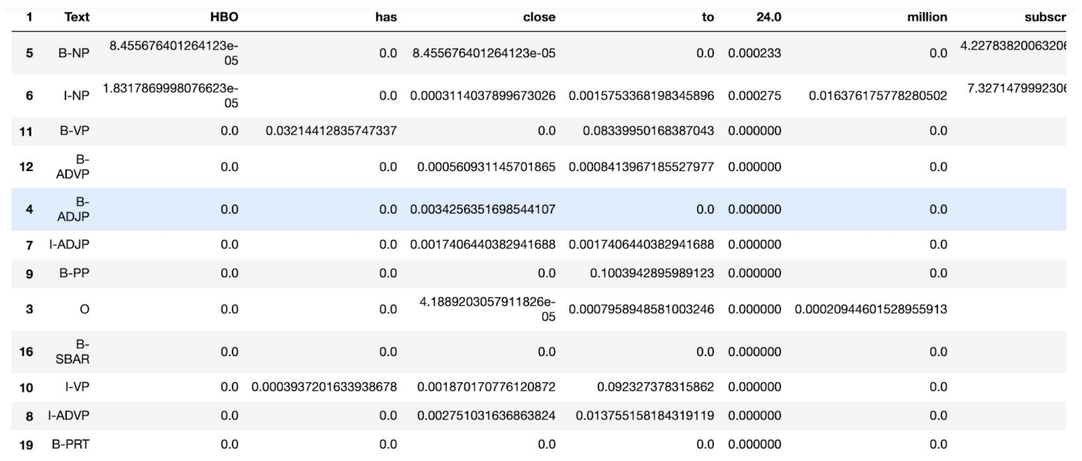
CN



```
#Entity in gold data: 700
#Entity in prediction: 4248

#Correct Entity : 345
Entity  precision: 0.0812
Entity  recall: 0.4929
Entity  F: 0.1395

#Correct Sentiment : 167
Sentiment  precision: 0.0393
Sentiment  recall: 0.2386
Sentiment  F: 0.0675
```

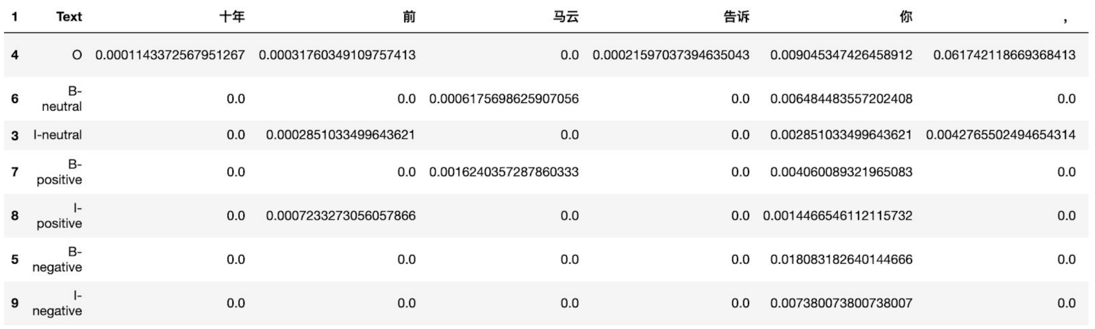
Part 3:

We started off by reusing part 2 processed data with a few edits on the data frame as we realized that data frame processing was too slow, we switched the data structure to a numpy array. The idea is to map each word (not unique) from the processed data in part 2 as a column to a numpy array and the row index of the numpy array dictates the unique labels of each word. We then exported the numpy array as a CSV which we will then load in our vertibi algorithm. The original csv looks like the following and needs some cleaning up:



1	Text	HBO	has	close	to	24.0	million	subscr
5	B-NP	8.455676401264123e-05	0.0	8.455676401264123e-05	0.0	0.000233	0.0	4.22783820063206
6	I-NP	1.8317869998076623e-05	0.0	0.0003114037899673026	0.0015753368198345896	0.000275	0.016376175778280502	7.3271479992306
11	B-VP	0.0	0.03214412835747337	0.0	0.08339950168387043	0.000000	0.0	
12	B-ADVP	0.0	0.0	0.000560931145701865	0.0008413967185527977	0.000000	0.0	
4	B-ADJP	0.0	0.0	0.0034256351698544107	0.0	0.000000	0.0	
7	I-ADJP	0.0	0.0	0.0017406440382941688	0.0017406440382941688	0.000000	0.0	
9	B-PP	0.0	0.0	0.0	0.1003942895989123	0.000000	0.0	
3	O	0.0	0.0	4.1889203057911826e-05	0.0007958948581003246	0.000000	0.00020944601528955913	
16	B-SBAR	0.0	0.0	0.0	0.0	0.000000	0.0	
10	I-VP	0.0	0.0003937201633938678	0.001870170776120872	0.092327378315862	0.000000	0.0	
8	I-ADVP	0.0	0.0	0.002751031636863824	0.013755158184319119	0.000000	0.0	
19	B-PRT	0.0	0.0	0.0	0.0	0.000000	0.0	

We transpose the received CSV and remove the headers and indexes, and converted it into a numpy array to represent the emission table. We also reindexed the rows such that it is has the same sequence as the dictionary of labels:



1	Text	十年	前	马云	告诉	你	，
4	O	0.0001143372567951267	0.00031760349109757413	0.0	0.00021597037394635043	0.009045347426458912	0.061742118669368413
6	B-neutral	0.0	0.0	0.0006175698625907056	0.0	0.006484483557202408	0.0
3	I-neutral	0.0	0.0002851033499643621	0.0	0.0	0.002851033499643621	0.0042765502494654314
7	B-positive	0.0	0.0	0.0016240357287860333	0.0	0.004060089321965083	0.0
8	I-positive	0.0	0.0007233273056057866	0.0	0.0	0.0014466546112115732	0.0
5	B-negative	0.0	0.0	0.0	0.0	0.018083182640144666	0.0
9	I-negative	0.0	0.0	0.0	0.0	0.007380073800738007	0.0

(This example, the 7 rows stands for the different labels and the columns indicates the words of the index)

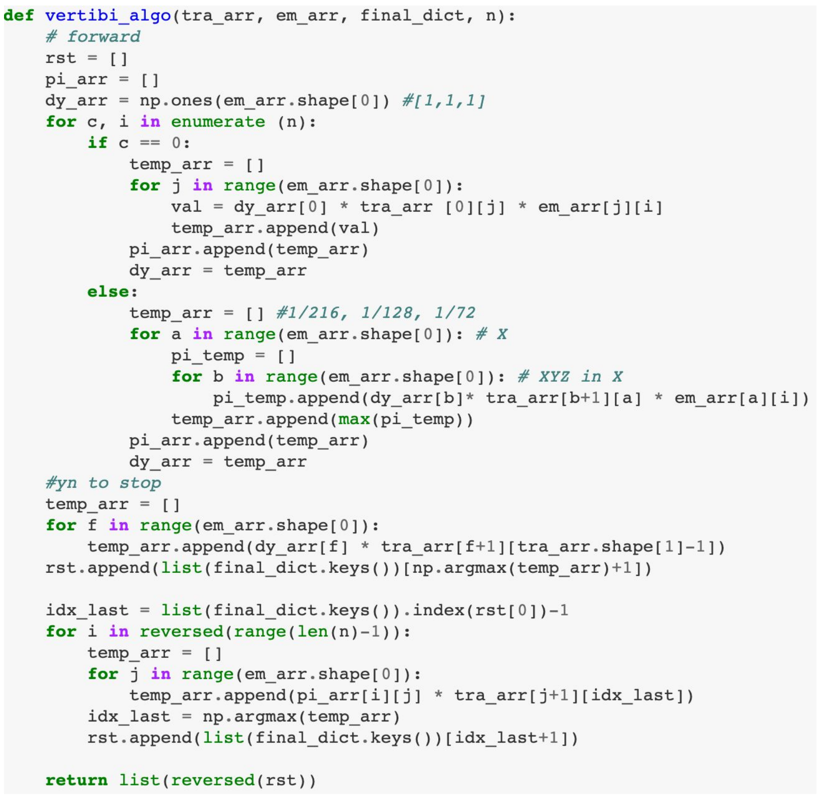
3.1

```
def transition_table(text, tra_arr):
    for ct, t in enumerate(text):
        for ci, i in enumerate(t):
            if ci == len(t)-1:
                tra_psn = list(final_dict.keys()).index(i[1])
                tra_arr[tra_psn][len(final_dict)-1] += 1/final_dict[i[1]]
            else:
                if ci == 0:
                    tra_psn = list(final_dict.keys()).index(i[1])
                    tra_arr[0][tra_psn-1] += 1/final_dict['START']
                cur_psn = list(final_dict.keys()).index(i[1])
                next_psn = list(final_dict.keys()).index(t[ci+1][1])
                tra_arr[cur_psn][next_psn-1] += 1/final_dict[i[1]]
    return tra_arr
```

Following the emission table, we have created a transition table (tra_arr) that will aid in our verbiti algorithm. The above algorithm maps the value of the combined transitions from e.g O -> B-positive for all data. To verify the transition table, each row is summed up to a value of 1. The dimension of the resulting transition table is of number of unique labels + 1 * number of unique labels + 1. The additional 1 accounts for the START and STOP state. The following is a mock-up table of the transition table:

	X	Y	Z	STOP
START	3/6	0	3/6	0
X	1/6	0	4/6	1/6
Y	1/4	0	0	3/4
Z	1/8	4/8	1/8	2/8

3.2



```
def vertibi_algo(tra_arr, em_arr, final_dict, n):
    # forward
    rst = []
    pi_arr = []
    dy_arr = np.ones(em_arr.shape[0]) #[1,1,1]
    for c, i in enumerate (n):
        if c == 0:
            temp_arr = []
            for j in range(em_arr.shape[0]):
                val = dy_arr[0] * tra_arr [0][j] * em_arr[j][i]
                temp_arr.append(val)
            pi_arr.append(temp_arr)
            dy_arr = temp_arr
        else:
            temp_arr = [] #1/216, 1/128, 1/72
            for a in range(em_arr.shape[0]): # X
                pi_temp = []
                for b in range(em_arr.shape[0]): # XYZ in X
                    pi_temp.append(dy_arr[b]* tra_arr[b+1][a] * em_arr[a][i])
                temp_arr.append(max(pi_temp))
            pi_arr.append(temp_arr)
            dy_arr = temp_arr
    #yn to stop
    temp_arr = []
    for f in range(em_arr.shape[0]):
        temp_arr.append(dy_arr[f] * tra_arr[f+1][tra_arr.shape[1]-1])
    rst.append(list(final_dict.keys())[np.argmax(temp_arr)+1])

    idx_last = list(final_dict.keys()).index(rst[0])-1
    for i in reversed(range(len(n)-1)):
        temp_arr = []
        for j in range(em_arr.shape[0]):
            temp_arr.append(pi_arr[i][j] * tra_arr[j+1][idx_last])
        idx_last = np.argmax(temp_arr)
        rst.append(list(final_dict.keys())[idx_last+1])

    return list(reversed(rst))
```

The verbiti algorithm takes in the following arguments:
Tra_arr: transmission array
Em_arr: emission array
final_dict: dictionary of unique labels
n: an array of number of paragraphs * number of words in the paragraph

The emission array is modified to only unique words. Instead of dealing with dictionaries, we deal with indexes as the values can be quickly obtained from the transition array and emission array.

The base case of the verbiti algorithm assumes $\pi(0, v) = 1 \ if \ v = START$.

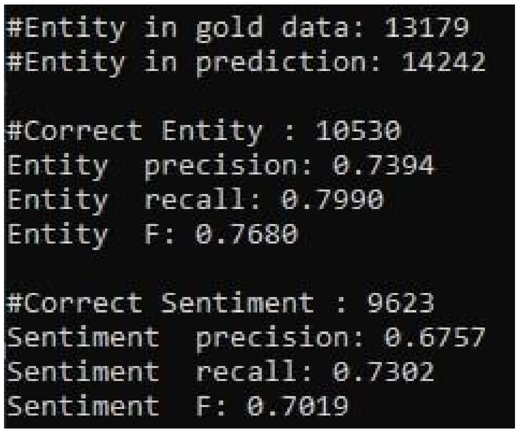
For $k \in number\ of\ words\ in\ a\ paragraph$, we will calculate the $\pi(k,\ v \in labels.a_{n,x}.b_{x}(d))$ and store the results of each π_{k} in a temporary array which will help us find the value of argmax quicker when doing back tracking.

For y_k to stop, calculate $\pi(k-1,\ STOP)$

For backtracking, we will calculate for $n \in k-2...1$:
$y^*_n = argmax\ \{\pi(k-2,\ v \in\ labels).\ a_v.STOP\ \}$. This will result in the most optimal sequence. We run each paragraph through the verbiti algorithm in which the results of each paragraph is being output to dev.out

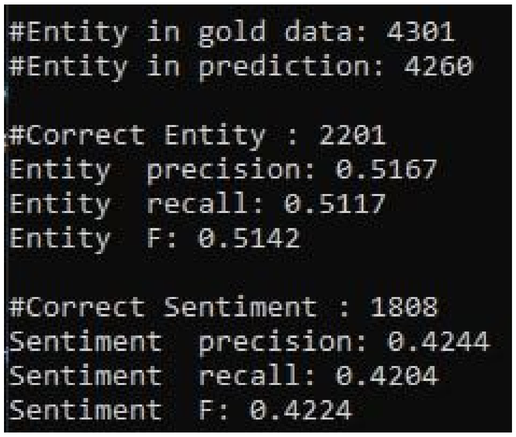
Part 3 Results

EN



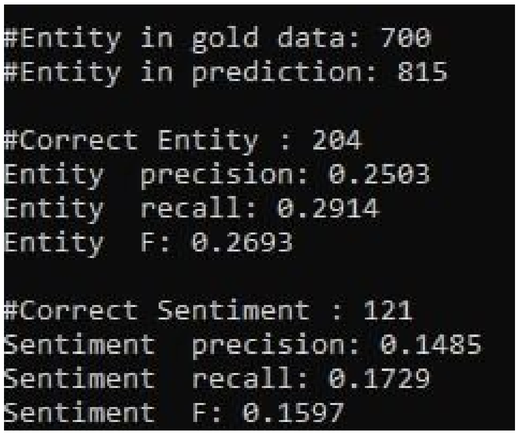
```
#Entity in gold data: 13179
#Entity in prediction: 14242

#Correct Entity : 10530
Entity  precision: 0.7394
Entity  recall: 0.7990
Entity  F: 0.7680

#Correct Sentiment : 9623
Sentiment  precision: 0.6757
Sentiment  recall: 0.7302
Sentiment  F: 0.7019
```

SG



```
#Entity in gold data: 4301
#Entity in prediction: 4260

#Correct Entity : 2201
Entity  precision: 0.5167
Entity  recall: 0.5117
Entity  F: 0.5142

#Correct Sentiment : 1808
Sentiment  precision: 0.4244
Sentiment  recall: 0.4204
Sentiment  F: 0.4224
```

CN



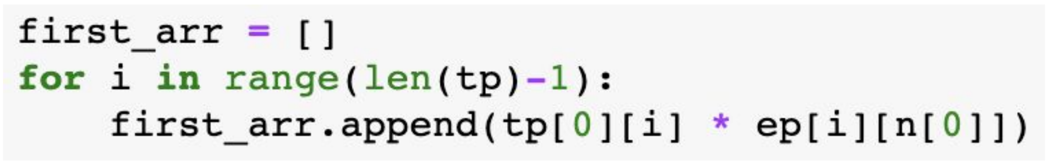
```
#Entity in gold data: 700
#Entity in prediction: 815

#Correct Entity : 204
Entity  precision: 0.2503
Entity  recall: 0.2914
Entity  F: 0.2693

#Correct Sentiment : 121
Sentiment  precision: 0.1485
Sentiment  recall: 0.1729
Sentiment  F: 0.1597
```

Part 4:

Part 4 loads the data as a numpy array that had been processed in part 3 as a .csv file.

4.1
Part 4 uses the same transition and emission parameters as part 3, but with a change in the Verbiti algorithm. To obtain the third best sequence for the sentence, we have to store the top 3 best scores in each node, instead of just the best score as we did in the normal Verbiti algorithm.

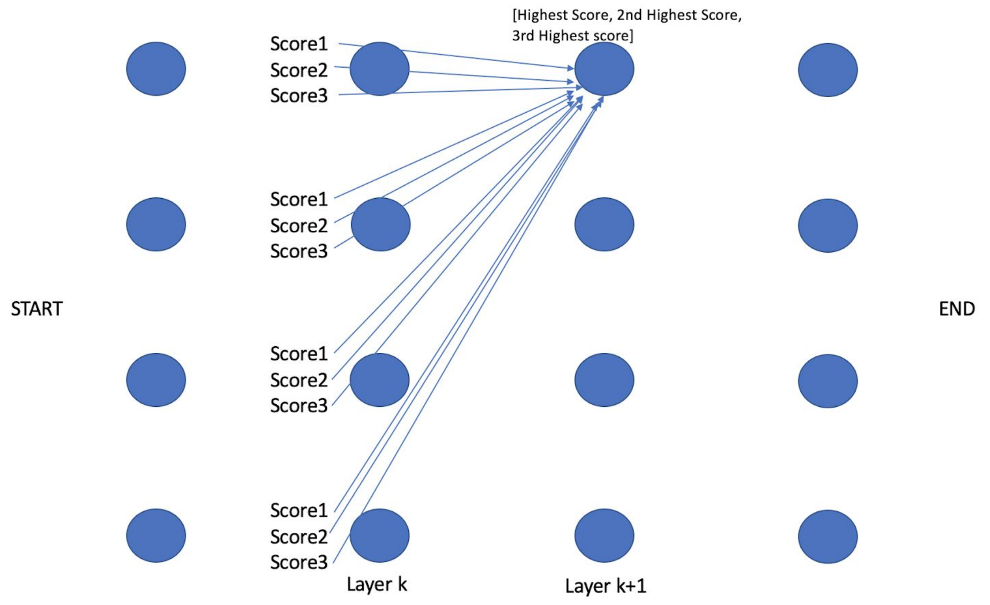
For the first layer,

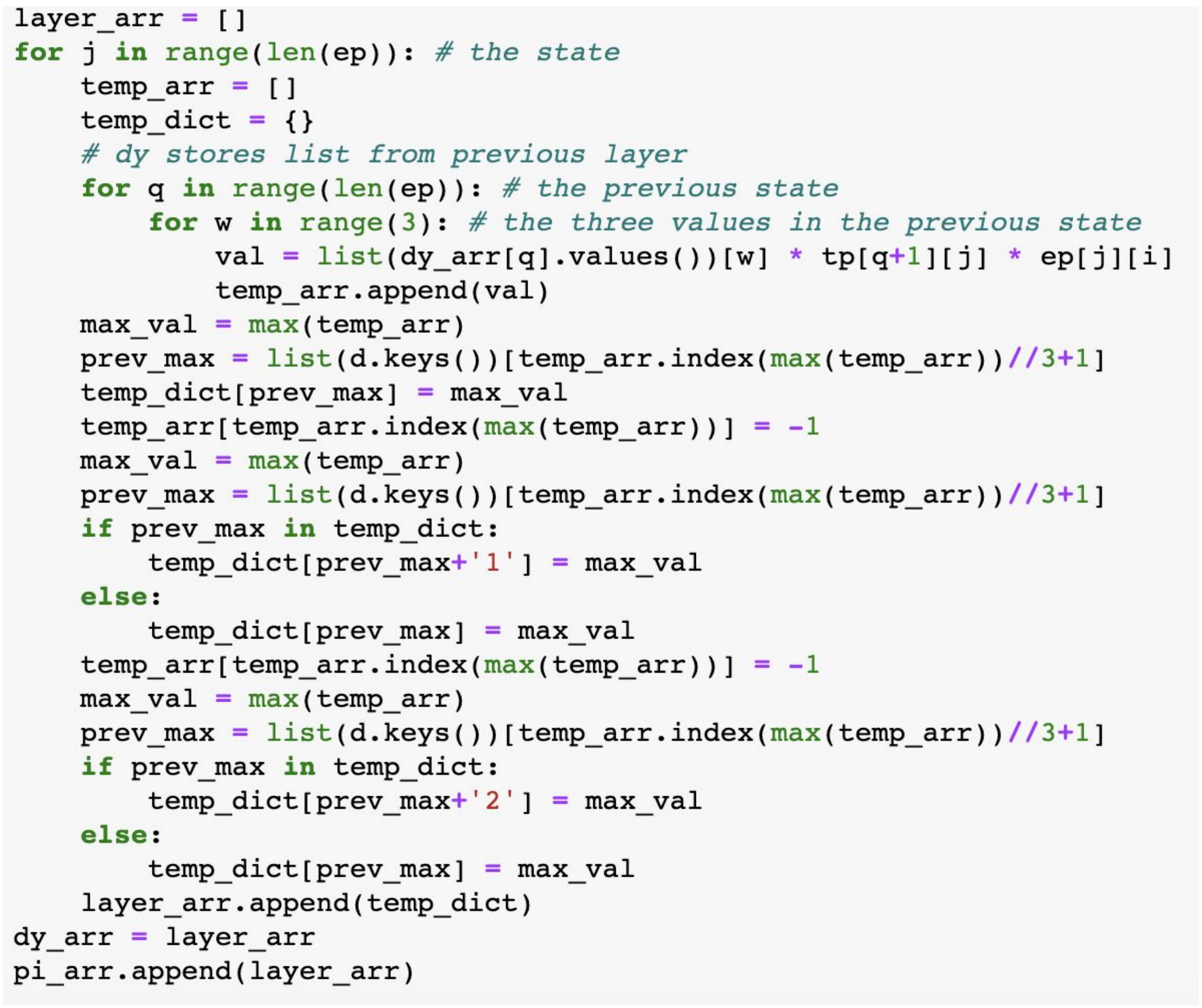


```
first_arr = []
for i in range(len(tp)-1):
    first_arr.append(tp[0][i] * ep[i][n[0]])
```

We simply get the scores from START to each node.

For the second layer onwards,

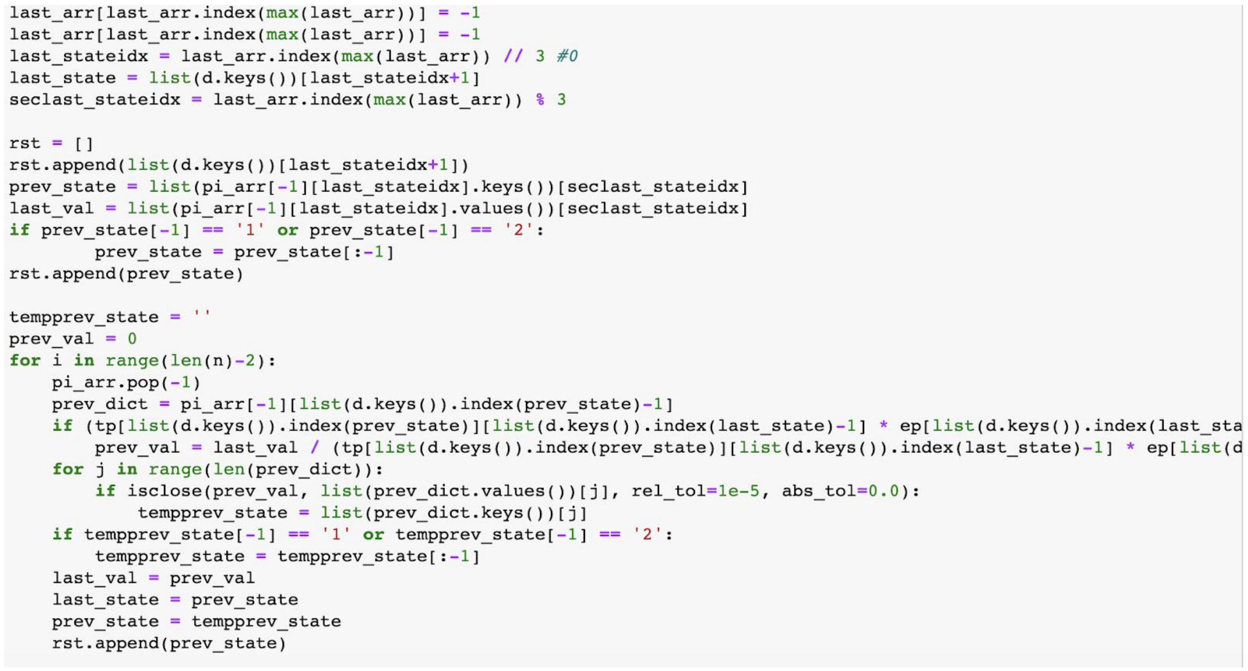




```
layer_arr = []
for j in range(len(ep)): # the state
    temp_arr = []
    temp_dict = {}
    # dy stores list from previous layer
    for q in range(len(ep)): # the previous state
        for w in range(3): # the three values in the previous state
            val = list(dy_arr[q].values())[w] * tp[q+1][j] * ep[j][i]
            temp_arr.append(val)
    max_val = max(temp_arr)
    prev_max = list(d.keys())[temp_arr.index(max(temp_arr))//3+1]
    temp_dict[prev_max] = max_val
    temp_arr[temp_arr.index(max(temp_arr))] = -1
    max_val = max(temp_arr)
    prev_max = list(d.keys())[temp_arr.index(max(temp_arr))//3+1]
    if prev_max in temp_dict:
        temp_dict[prev_max+'1'] = max_val
    else:
        temp_dict[prev_max] = max_val
    temp_arr[temp_arr.index(max(temp_arr))] = -1
    max_val = max(temp_arr)
    prev_max = list(d.keys())[temp_arr.index(max(temp_arr))//3+1]
    if prev_max in temp_dict:
        temp_dict[prev_max+'2'] = max_val
    else:
        temp_dict[prev_max] = max_val
    layer_arr.append(temp_dict)
dy_arr = layer_arr
pi_arr.append(layer_arr)
```

We consider the path from each node in the previous layer to each node in the next layer. For each node of the next layer, there would be 3x(number of nodes) different values considered, and the top 3 of all these values would be stored.

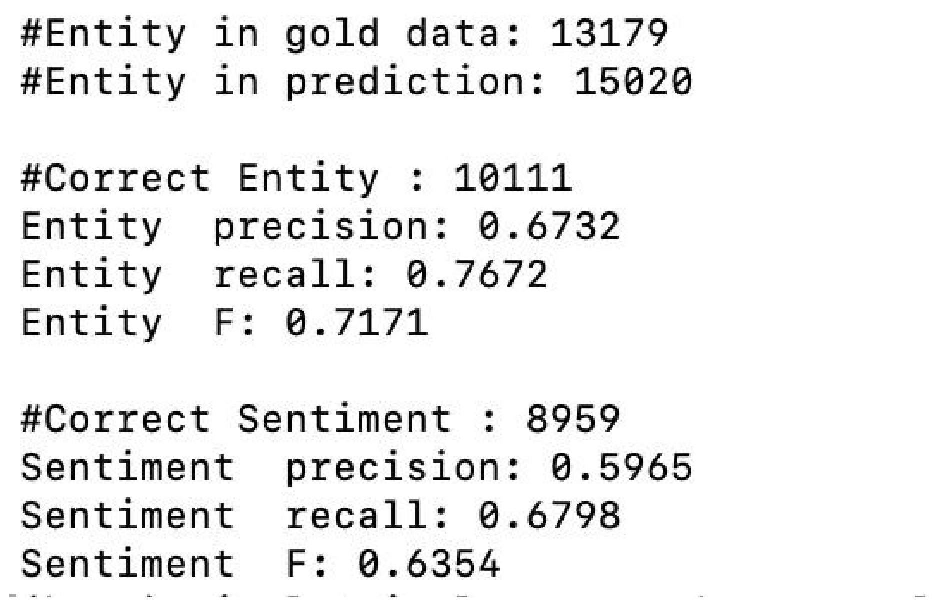
When backtracking, we need to find out which of the three values stored actually leads to the path of the third best value. We will know the last node, and the previous node that lead to it. We know this through the index of the argmax, as we did in part 3. Knowing the score in the node of the last layer that is in the third best path, we determine the previous score by dividing that score by the transition and emission parameters of the previous node. We then match this score to one of the three scores stored in the previous node.



```
last_arr[last_arr.index(max(last_arr))] = -1
last_arr[last_arr.index(max(last_arr))] = -1
last_stateidx = last_arr.index(max(last_arr)) // 3 #0
last_state = list(d.keys())[last_stateidx+1]
seclast_stateidx = last_arr.index(max(last_arr)) % 3

rst = []
rst.append(list(d.keys())[last_stateidx+1])
prev_state = list(pi_arr[-1][last_stateidx].keys())[seclast_stateidx]
last_val = list(pi_arr[-1][last_stateidx].values())[seclast_stateidx]
if prev_state[-1] == '1' or prev_state[-1] == '2':
        prev_state = prev_state[:-1]
rst.append(prev_state)

tempprev_state = ''
prev_val = 0
for i in range(len(n)-2):
    pi_arr.pop(-1)
    prev_dict = pi_arr[-1][list(d.keys()).index(prev_state)-1]
    if (tp[list(d.keys()).index(prev_state)][list(d.keys()).index(last_state)-1] * ep[list(d.keys()).index(last_sta
        prev_val = last_val / (tp[list(d.keys()).index(prev_state)][list(d.keys()).index(last_state)-1] * ep[list(d
    for j in range(len(prev_dict)):
        if isclose(prev_val, list(prev_dict.values())[j], rel_tol=1e-5, abs_tol=0.0):
            tempprev_state = list(prev_dict.keys())[j]
    if tempprev_state[-1] == '1' or tempprev_state[-1] == '2':
        tempprev_state = tempprev_state[:-1]
    last_val = prev_val
    last_state = prev_state
    prev_state = tempprev_state
    rst.append(prev_state)
```

Part 4 Results

EN



```
#Entity in gold data: 13179
#Entity in prediction: 15020

#Correct Entity : 10111
Entity  precision: 0.6732
Entity  recall: 0.7672
Entity  F: 0.7171

#Correct Sentiment : 8959
Sentiment  precision: 0.5965
Sentiment  recall: 0.6798
Sentiment  F: 0.6354
```

Part 5:

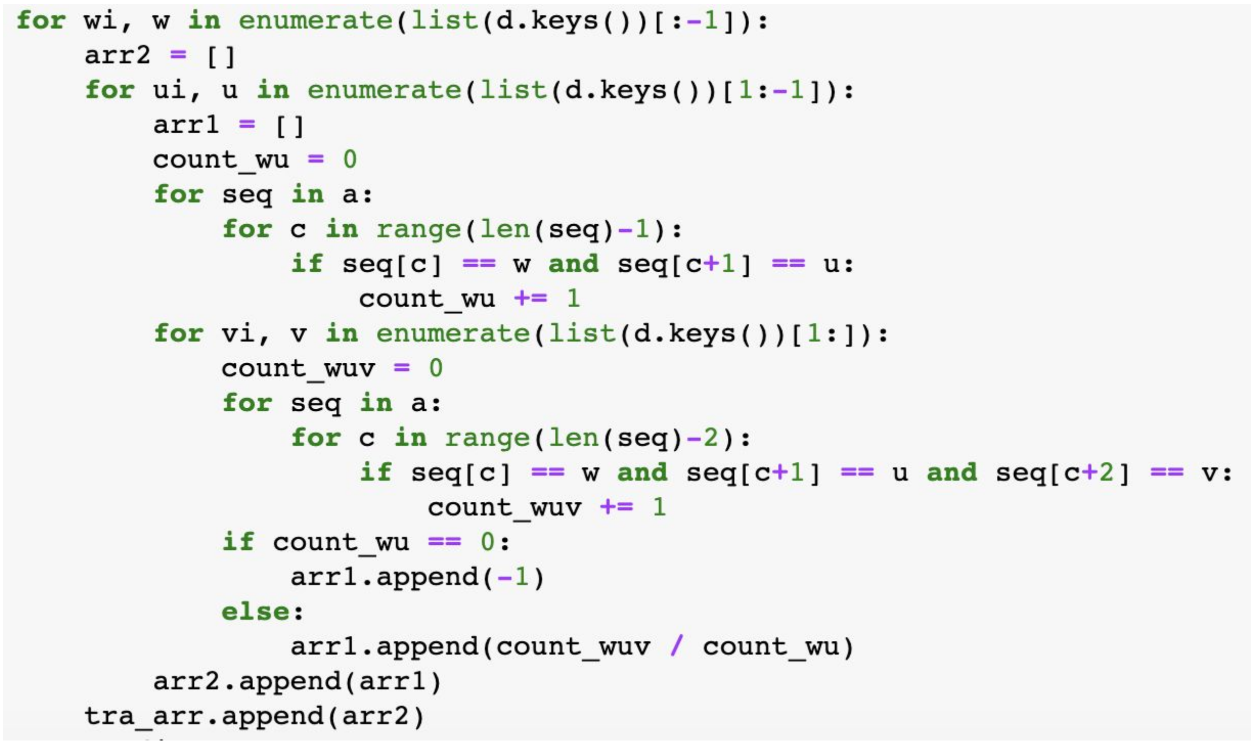
Part 5 loads the data as a numpy array that had been processed in part 3 as a .csv file.

Part 5 uses a 2nd order HMM.

The emission parameters remain the same as part 3, as each tag still leads to the same observation.

As for the transition parameters, we have to consider the two tags before the current tag. Our transition array is a 3D array. Given n tags, the size of our array is (n+1)x(n)x(n+1). The +1 accounts for both the START and STOP states.
For an example with three states, tra_arr: [[[START,X],[START,Y],[START,Z]], [[X,X],[X,Y],[X,Z]].....]

- [START,X] contains 4 values: (START->X->X), (START->X->Y), (START->X->Z), (START->X->STOP)



```
for wi, w in enumerate(list(d.keys())[:-1]):
    arr2 = []
    for ui, u in enumerate(list(d.keys())[1:-1]):
        arr1 = []
        count_wu = 0
        for seq in a:
            for c in range(len(seq)-1):
                if seq[c] == w and seq[c+1] == u:
                    count_wu += 1
        for vi, v in enumerate(list(d.keys())[1:]):
            count_wuv = 0
            for seq in a:
                for c in range(len(seq)-2):
                    if seq[c] == w and seq[c+1] == u and seq[c+2] == v:
                        count_wuv += 1
            if count_wu == 0:
                arr1.append(-1)
            else:
                arr1.append(count_wuv / count_wu)
        arr2.append(arr1)
    tra_arr.append(arr2)
```

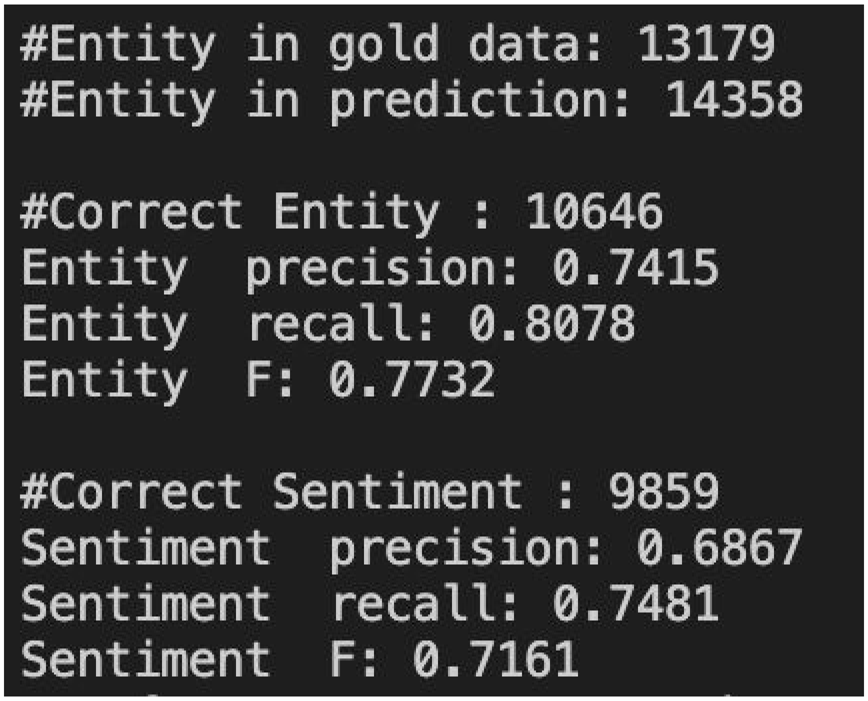
For each a(w,u,v), where the current node is at state v, we calculate the count of the previous node being u, and the node before that being w as count(w,u,v). We then calculate count(w,u) as the number of times state w occurred right before state u. a(w,u,v) = count(w,u,v) / count(w,u)
For the verbiti algorithm, we imagine there to be a node before start, which we label as -1. So for each node of the first layer, we calculate a(-1,START,v).

The rest of the Verbiti Algorithm follows the same logic as part 3, but with the transition parameters replaced.

Part 5 obtains better results than part 3 because instead of simply considering the previous tag, it considers the two precious tags. This allows the model to learn the pattern of sequences more accurately.

Part 5 Results

EN



```
#Entity in gold data: 13179
#Entity in prediction: 14358

#Correct Entity : 10646
Entity  precision: 0.7415
Entity  recall: 0.8078
Entity  F: 0.7732

#Correct Sentiment : 9859
Sentiment  precision: 0.6867
Sentiment  recall: 0.7481
Sentiment  F: 0.7161
```