50.007 Project

Gargi Pandkar (1004680), Mihir Chhiber (1004359)

December 12, 2021

1 Part 1

To estimate the emission parameters from the training set, we used maximum likelihood estimation (MLE). In our implementation, we defined two dictionaries *e_count* and *e_prob* - one to track the counts of each state and another to store the count of word given the state which is later divided using the first dictionary to give us emission probability. We increase the count of each state by 1 when dividing to help account for words which do not appear in the training set but may appear in the test set. Therefore

```
e_prob[y]["#UNK#"] = 1/e_count[y] + k
e_prob[y][x] = e_prob[y][x]/e_count[y] + k
```

We replace x with UNK if x is not observed in the training set, therefore 0 is not returned which helps improve computation in part 2, 3 and 4. We store the counts for each state.

```
e_count = {'O': 31627, 'B-positive': 1274, 'B-neutral': 85, 'B-negative': 429,
           'I-positive': 400, 'I-neutral': 44, 'I-negative': 229}
```

Also we store the count for corresponding input word and state and to get probability these are divided by the state counts.

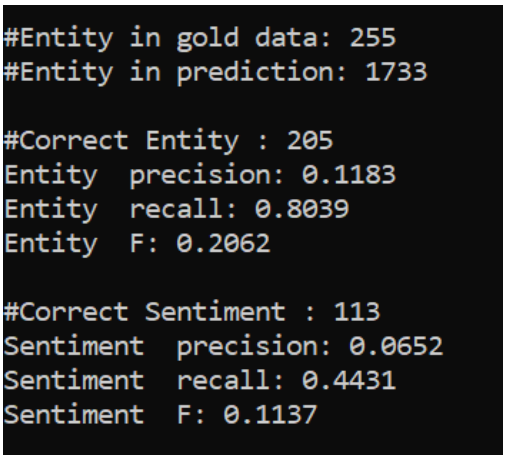
```
e_prob = {'O': {'disfrutemos': 3.161755406601745e-05, 'de':  ...},
       'B-positive': {'producto': 0.009411764705882352, 'ambiente':  ...}
```

To predict the state of the input sentence using the emission parameter, we used the formula given below where we take the argument y (represents state) with the largest emission probability for the input word.

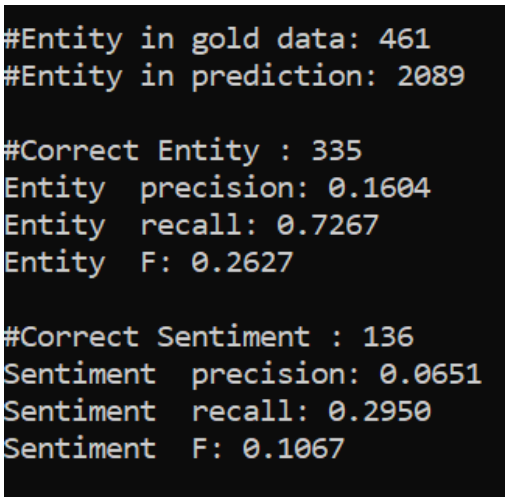
$$y^* = \arg\max_y e(x|y)$$

We used *evalResult.py* to evaluate the performance of our prediction as reported below.

1.1 Results



(a) ES



(b) RU

Figure 1: Results for Part 1 (a) ES (b) RU

2 Part 2

2.1 Approach

The emission parameters are estimated from the training set via the same function as in Part 1.

The transition parameters are estimated from the training set using maximum likelihood estimation (MLE). Within the function we keep two dictionaries - one to track the counts of each transition source state and another to store the transition probabilities.

A two-level nested dictionary is used to store the transition probabilities. So that $q\_prob[i][j]$ will give the transition probability from state i to state j. If a transition is not observed, including the cases $(y_i \rightarrow START)$ and $(STOP \rightarrow y_i)$, there is no record for it and instead the default value of 0 or min_num is returned when retrieving from the dictionary.

```
pie = {'START': {'O': 0.8487074030552292, 'B-positive': 0.1113396004700 ...},
       'O': {'O': 0.8730958781362007, 'STOP': 0.0761872759856 ...}, ...}
```

2.2 Viterbi algorithm

A two-level nested dictionary is used to store the scores (i.e. probabilities of a given state at a given timestep). The first-level keys are the timesteps/sequence positions and the second-level keys are the states (including START and STOP).

```
pie = {0: {'START': 1, 'O': min_num, 'B-positive':min_num, ....}}
```

Another dictionary is used to store the state corresponding to the highest score at every timestep. This saves us calculations during backtracking as we can simply read off the path from this dictionary.

$$y_1^*, ..., y_n^* = \arg\max_{y_1,...,y_n} p(x_1, ..., x_n, y_1, ..., y_n)$$

Initialization
We have two dictionaries pie and path where pie accounts for the score of the best path based on the state of the i-th input word , and path accounts for the best $i-1$-th word's state using the input word and its temporary assigned state. We initialise the values for the initial states where *pie* value for $START$ is 1 and min_num for all the other states, forcing the algorithm to choose $START$ as the state to start with. We initialise the value in path as well.

Recursion
We use a while loop where we iterate through every word in the sentence from left to right. Inside the while loop, we have two nested for loops which calculate the i-th path score for i-th state and path's $i-1$-th state for every i-th state. This process is repeated until all the words are read. The score for each path are calculated by taking product of score of $i-1$-th path given the $i-1$-th state, transition probability of i-th state and $i-1$-th state, and emission probability of input word x given i-th state. The $i-1$-th state is selected based on the highest path score.

Termination and backtracing
After calculating the path and pie in the while loop, we find the nth state given $n+1$-th state is $STOP$. To calculate the path score, we only the the score of the nth path and transition probability of nth state and $n+1$-th state. Using the best n-th state, we backtrack using path dictionary to find the $n-1$-th state and repeat the process to get the optimal path of states for the given input.

2.3 Results

```
#Entity in gold data: 255
#Entity in prediction: 551

#Correct Entity : 131
Entity  precision: 0.2377
Entity  recall: 0.5137
Entity  F: 0.3251

#Correct Sentiment : 104
Sentiment  precision: 0.1887
Sentiment  recall: 0.4078
Sentiment  F: 0.2581
```

(a) ES

```
#Entity in gold data: 461
#Entity in prediction: 534

#Correct Entity : 219
Entity  precision: 0.4101
Entity  recall: 0.4751
Entity  F: 0.4402

#Correct Sentiment : 144
Sentiment  precision: 0.2697
Sentiment  recall: 0.3124
Sentiment  F: 0.2894
```

(b) RU

Figure 2: Results for Part 2 (a) ES (b) RU

3 Part 3

3.1 Approach

The approach taken in top-k HMM is similar to the approach taken in part 2. We calculate the best k paths for every word in the sentence and use the k paths for the next word to find next k optimal paths. In this implementation, we need to keep track of the state of the word and the corresponding path taken for the state as the state can have k paths. Our function $train\_part3$ takes in argument top_k which is set to 1 by default and we report results for $top\_k = 5$.

This leads to adding another for loop in the nested for loops to keep track of best k paths for the i-th word given the i-th state. We had to change dictionaries *pie* and *path* where instead of single state and path score, k states with the specific path number and k path scores were stored. Therefore the time complexity of the algorithm is $O(nk^2|T|^2)$.

```
while i<len(sentence)+1:
  pie[i] = {}
  path[i] = {}
  # current state
  for u in state_ls:
    max_path = random.choice(state_ls)
    # previous state
    pie[i][u] = [-1]*top_k
    path[i][u] = ['']*top_k
    for v in state_ls+['START']:
      for r in range(top_k):
        prev_p = pie[i-1].get(v, [0]*top_k)[r]
        q = q_prob[v].get(u,0)
        e = e_prob[u].get(sentence[i-1],e_prob[u]["#UNK#"])
        p = prev_p * q * e
        for a in range(top_k):
          if p > pie[i].get(u,0)[a]:
            if not ([v,r] in path[i][u]):
              pie[i][u][a+1:] = pie[i][u][a:-1]
              pie[i][u][a] = p
              path[i][u][a+1:] = path[i][u][a:-1]
              path[i][u][a] = [v,r]
              break
```

Backtracking similar to part 2 is followed but the loop finds k paths. We return the $k-1$-th path.

3.2 Results

```
#Entity in gold data: 255
#Entity in prediction: 528

#Correct Entity : 108
Entity  precision: 0.2045
Entity  recall: 0.4235
Entity  F: 0.2759

#Correct Sentiment : 73
Sentiment  precision: 0.1383
Sentiment  recall: 0.2863
Sentiment  F: 0.1865
```

(a) ES

```
#Entity in gold data: 461
#Entity in prediction: 751

#Correct Entity : 201
Entity  precision: 0.2676
Entity  recall: 0.4360
Entity  F: 0.3317

#Correct Sentiment : 108
Sentiment  precision: 0.1438
Sentiment  recall: 0.2343
Sentiment  F: 0.1782
```

(b) RU

Figure 3: Results for Part 3 (a) ES (b) RU

4 Part 4

4.1 Approach

We chose to implement a second-order Hidden Markov Model for the design challenge. We also attempt to improve handling of unseen words (i.e. words that do not appear in the training set). Second-order HMM is based on a similar principle as the first-order HMM, but two previous states are considered during expectation maximization.

4.2 Parameter Matrix Training

The emission parameters are estimated from the training set using maximum likelihood estimation (MLE). However, the emission parameters for unseen words are calculated differently. We refer to the methods and results from *Haulrich*, 2009. Any unseen word is processed as the UNK token and the emission parameters for this token are determined by overall tag distribution.

$$e(UNK|y_i) = \frac{\text{Count}(y_i)}{\sum_{y_i \in \{T\}} Count(y_i)}$$

In the given training data, tag 'O' occurs with large frequency and hence the distribution is skewed. We did not see much improvement from the naive smoothing method.

The transition parameters are estimated from the training set using a modified version of the function in Part 2. We now need to map the likelihood of tag/state pairs that lead to the next tag/state. Aside from the additional $START$ and $STOP$ states, we also have a $PRESTART$ state to represent the first transition source as $(PRESTART, START)$.

$$q(y_i + 1|y_i, y_i - 1) = \frac{\text{Count}(y_i + 1, y_i, y_i - 1)}{\text{Count}(y_i, y_i - 1)}$$

4.3 Modified Viterbi Algorithm

The Viterbi algorithm is modified to work with tag/state pairs as the source for a transition. Both the score and backpointer dictionaries use state pairs (tuples) as keys.

At each timestep of the algorithm, we consider a pair of previous states and to determine the next state we iterate over all states for each pair. There are n timesteps, T states and T^2 pairs. Hence, the time complexity of this modified algorithm is $O(n|T|^3)$.

Initialization
The score and backpointer dictionaries are indexed by tag/state pairs.

```
pie = { 0: {('PRESTART','START'):1, ..., ('I-negative','STOP'):-inf},
        1: { ... }, ... }
path = {0:('PRESTART', 'START')}
```

Recursion
The transmission parameters and previous score are retrieved as such:

```
q = 0 if q_prob.get((t, v), -1) == -1 else q_prob[(t, v)].get(u,0)
prev_score = pie[i-1].get((t, v), min_num)
```

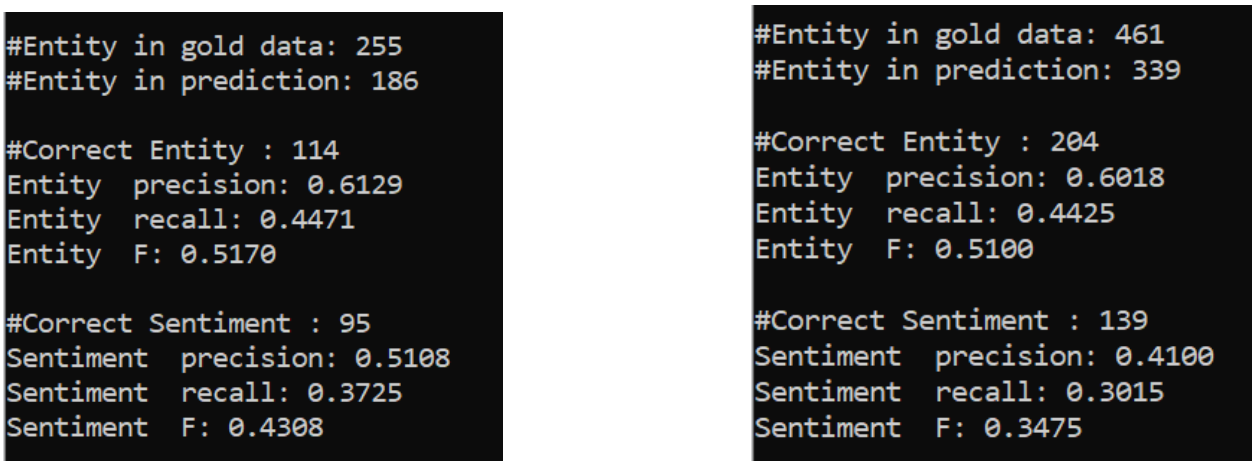
To handle arithmetic underflow we do sum over log terms instead of multiplying the terms, same as in Part 2. In our code, we refer to the states in order of timesteps (i.e. grandparent, parent, current) by the variables t, v and u. We now have a three-level nested loop.

```
while i<len(sentence)+1:
    pie[i] = {}
    path[i] = {}
    # current state
    for u in state_ls:
        max_path = 'O'
        # previous state
        for v in ['START']+state_ls:
            # 2nd previous state
            for t in ['PRESTART', 'START']+state_ls:
                #UPDATE RULE FOR SCORE AND PATH
```

Termination and backtracking
Termination follows same update rule as recursion with $u = STOP$ and for v and u in $state_ls$ except we don't consider any emission parameter. For obtaining best path, we can read off directly from dictionary again. Due to indexes being state pairs, at each timestep we form the index to read off from the previous timestep.

```
best_path = []
state_2seq = max(pie[i], key=pie[i].get)
while i!=0:
    best_path.append(state_2seq[0])
    state_2seq = (path[i][state_2seq], state_2seq[0])
    i-=1
best_path.reverse()
```

4.4 Results



(a) ES (b) RU

Figure 4: Results for Part 4 (a) ES (b) RU

References

Haulrich, M. (2009). Different Approaches to Unknown Words in a Hidden Markov Model Part-of-Speech Tagger.