50.007 Machine Learning Project

Madhumitha Balaji ISTD, 1004471
Shyam Sridhar ESD, 1004360
Lawrence Leung Chew ISTD, 1004442

December 2021

Introduction

This report details the approaches and results of our design project in which we aim to design a sequence labelling model for informal texts using the Hidden Markov Model (HMM) that we have learned in class. The files for this project were provided and located in RU folder and ES folder. For each dataset, there is a labelled training set train, an unlabelled development set dev.in, and a labelled development set dev.out. The labelled data has the format of one token per line with token and tag separated by tab and a single empty line that separates sentences. Overall, our goal is to build a sequence labelling system from such training data and then use the system to predict tag sequences for new sentences.

1 Part 1

First, we created a python file **utils.py** containing helper functions to load and preprocess the data given:

1. **data_dump(path)** will return a list sequence [['sentence1_word1 label1', 'sentence1_word2 label2', 'sentence1_word3 label3'], ['sentence2_word1 label1']] for each sentence after reading the file on path. **data_dump_shuffle(path)** is the same but the output is shuffled
2. **data_dump_split(path)** is similar to data_dump(path) but will return list sequence [[word_1,label_1],[word_2,label_2].....[word_n,label_n]...] for each sentence.
3. **data_dump_split_test(path)** has the same functionality as **data_dump_split(path)** but applied on the dev/test datasets.

1.1 Finding Emission parameters

To estimate the emission parameters, we first defined the function **emission_counting(path)** to return the number of observations for each word in a line emitted by a specific

state and a nested dictionary that keeps track of the number of observations per specific word or observation in which the outer key is the specific state or tag. The nested dictionary is in this format: { key = state : {key = observation : value = count }}

This function would hence allow us to more easily retrieve values necessary to calculate emission parameters as given by the equation below:

$$e(x|y) = \frac{\texttt{Count}(y \rightarrow x)}{\texttt{Count}(y)}$$

Considering our dictionary, we can find the numerator for emission parameter by looking for **emission_dict [y][x]** and afterwards find the denominator by calculating **sum(emission_dict[y].values())** We hence implemented the function **estimate_emission** for this purpose.

Next, we needed to modify that function to cater for words that appear in the test set but do not appear in the training set. Hence, when a special token **#UNK#** is assigned for such word, we will set the numerator to k. We then find the emission parameter by following the equation provided below:

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

The function **estimate_emission_param** uses a if-else statement to check if the word is indeed found in the training set and assign the special token if it is not found. The value of k is set as such: k = 1 to allow for calculation of emission parameters.

Another function **labelling** was then created to take as input a sentence as a list with each word as a string and return a list of predicted labels (y) for that sentence.

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

We loop through all the states of each word in a sentence and figure out their respective emission probabilities. We would hence return the state with the highest emission probability as predicted label y and return a list of all predicted labels.

1.2 Results for Part 1

We used the **evalResult.py** to calculate precision, recall and F score. Our results for part 1 are:

```
#Entity in gold data: 255
#Entity in prediction: 1733

#Correct Entity : 205
Entity  precision: 0.1183
Entity  recall: 0.8039
Entity  F: 0.2062

#Correct Sentiment : 113
Sentiment  precision: 0.0652
Sentiment  recall: 0.4431
Sentiment  F: 0.1137
```

Figure 1: Results for ES part 1

```
#Entity in gold data: 461
#Entity in prediction: 2098

#Correct Entity : 335
Entity  precision: 0.1597
Entity  recall: 0.7267
Entity  F: 0.2618

#Correct Sentiment : 136
Sentiment  precision: 0.0648
Sentiment  recall: 0.2950
Sentiment  F: 0.1063
```

Figure 2: Results for RU part 1

2 Part 2

For finding the transition parameters, we need to consider 2 additional cases i.e "START" and "STOP" states as compared to emission parameters.

2.1 Finding Transition parameters

For part 2 to estimate the transition parameters, we first wrote the function **transition_counting(path)** to return a nested dictionary that keeps track of the number of observations of transiting from one state to another per specific word or observation in which the outer key is the specific state or tag. The **transition_dict** is in the format: { key = state_i1 : { key = state_i2 : value = count} }

This function would hence allow us to more easily retrieve values necessary to calculate transition parameters as given by the equation below:

$$q(y_i|y_{i-1}) = \frac{\texttt{Count}(y_{i-1}, y_i)}{\texttt{Count}(y_{i-1})}$$

Considering our dictionary, we can find the numerator for transition parameters by looking for **transition_dict [state_i1][state_i2]** and afterwards find the denominator by calculating **sum(transition_dict[state_i1].values())** We hence implemented the function **get_transition_parameters** for this purpose.

2.2 Viterbi Algorithm

Using the estimated emission and transition parameters we calculated in previous parts we implemented the viterbi algorithm to compute the following:

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

First, we retrieve a list of all possible states with "START" and "STOP" removed). We keep 'observations' in the form returned by **data_dump** and initialize **pi_scores** to keep of the scores to each state from every previous state.

Our main function loops through each observation in the dataste. Whenever an unseen word is encountered, it is replaced with the **#UNK#** token. We obtain the transition and emission parameters using **transition_counts** and **emission_counts** dictionaries obtained from the **transition_counting** and **emission_counting** functions. To avoid underflow issue, we take logarithm of emission and transition parameters and in the case where the probability value is equal to zero, we set the log value to negative infinity. Since we used log, we will add pi_score + emission probability + transition probability (we do not multiply).

For the first state, pi_scores[0]['START'] = 1 and log(1) = 0 so we will ignore this term. When we reach the last word, we store the maximum score over each state

and perform the transition from last state to "STOP" similarly as mentioned before. NOTE: *no emission for "STOP" state.* Finally, we run through the backward algorithm at the end to obtain the ultimate path which is returned by the **viterbi** function.

2.3 Results for Part 2

Similar to part 1, we use the **evalResult.py** to calculate precision, recall and F score. Our results for part 1 are:

```
#Entity in gold data: 255
#Entity in prediction: 684

#Correct Entity : 133
Entity  precision: 0.1944
Entity  recall: 0.5216
Entity  F: 0.2833

#Correct Sentiment : 110
Sentiment  precision: 0.1608
Sentiment  recall: 0.4314
Sentiment  F: 0.2343
```

Figure 3: Results for ES part 2

```
#Entity in gold data: 461
#Entity in prediction: 1163

#Correct Entity : 253
Entity  precision: 0.2175
Entity  recall: 0.5488
Entity  F: 0.3116

#Correct Sentiment : 157
Sentiment  precision: 0.1350
Sentiment  recall: 0.3406
Sentiment  F: 0.1933
```

Figure 4: Results for RU part 2

3 Part 3

3.1 Find the 5-th best output sequence

Using the estimated transition and emission parameters, we implemented a modified Viterbi algorithm to find the 5-th best output sequences. We obtain the top best 5 paths and store these values. The algorithm that we implemented is as follows:

1. Initialization:
$$\pi(0, r_u) = \begin{cases} 0 & u = \text{START} \\ -\infty & \text{otherwise} \end{cases}$$

2. For $j = 0, 1, ...(n-1)$, for each $U \in T$ where route r_u is the total path leading up to and including u, calculating the top 5 probabilities and keeping a record of their paths
$$\pi(j+1, r_u) = topfive_v\{\pi(j, r_v) + \log(b_u(x_{j+1})) + \log(a_{v,u})\}$$

3. Retrieving the best 3 scores:
$$\pi(n+1, r_{STOP}) = topfive_v\{\pi(n, r_v) + \log(a_{v,STOP})\}$$

We store the 5 highest scores in the list findmax. We find the argument that gives the maximum value among all these scores and store then in a list state_ans. The format of our lists are: [highest score, 2nd highest score, ... , 5th highest score], and [highest state, 2nd highest state, ... , 5th highest state] respectively. Using our backward algorithm, we find the 5 best paths, by iterating over each node and keeping track of its previous node's 5 best paths and appending to it. **ultimate_path** is a list of 5 lists, that holds 5 best paths in decreasing order. On taking the last element, we can find the 5th best path.

3.2 Results for Part 3

For part 3, we use the **evalResult.py** again to calculate precision, recall and F score. Our results for part 3 are:

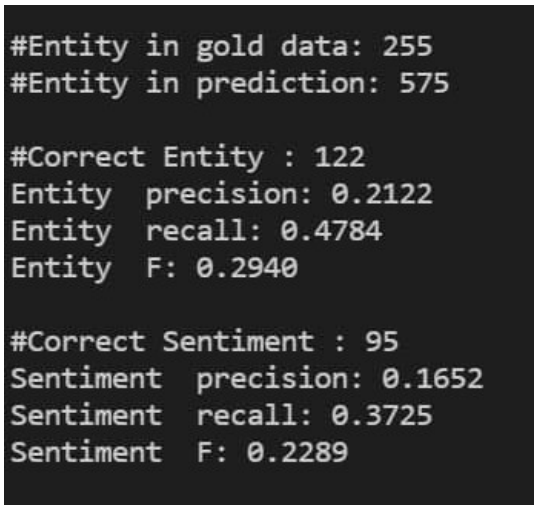

Figure 5: Results for ES part 3

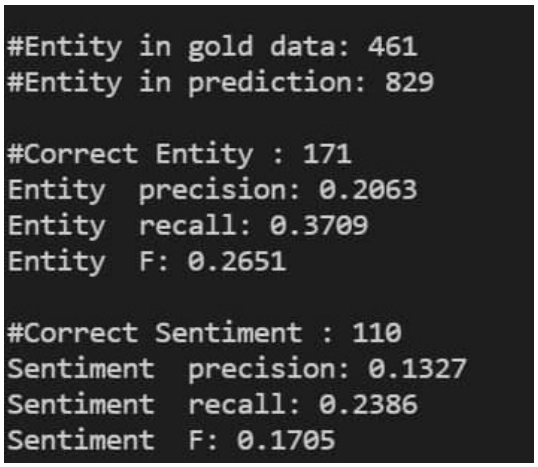

Figure 6: Results for RU part 3

4 Part 4

4.1 Design Challenge

For part 4, based on the training and development set, we developed a better design for an improved sentiment analysis system for tweets using the **structured perceptron algorithm**, which is outlines as follows.

Inputs: Training examples (x_k, y_k)
Initialization: $\overline{\lambda} = 0$
Algorithm:
  For $l = 1$ to L, $k = 1$ to n
    Use Viterbi to get $z_k = \text{argmax}_z \overline{\lambda} \cdot \Phi(x_k, z)$
    If $z_k \neq y_k$ then $\overline{\lambda} = \overline{\lambda} + \Phi(x_k, y_k) - \Phi(x_k, z_k)$
Output: weights $\overline{\lambda}$

ϕ is the global feature vector; λ is the corresponding weight vector

4.2 Our Implementation

We implemented a **generate_features** function, where we define the following features to represent a feature vector for our data.

```
features = [f"TAG_{tag}",
            f"TAG_2ORDER_{prev_tag}_{tag}",
            f"WORD_{observation}",
            f"WORD_BIGRAM_{prev_word}_{observation}",
            f"WORD_TRIGRAM_{prev_word}_{observation}_{next_word}",
            f"WORD_LOWER+TAG_{lower_case}_{tag}",
            f"UPPER_{observation[0].isupper()}_{tag}",
            f"PUNCTUATION_{observation in string.punctuation}",
            f"LOWERCASE+TAG_{lower_case}_{tag}",
            f"SUFFIX_3_{suffix_3}",
            f"PREFIX_3_{prefix_3}",
            f"SUFFIX_TAG_{suffix_3}_{tag}",
            f"SUFFIX_TAG_PREVTAG_{suffix_3}_{tag}_{prev_tag}",
            f"PREFIX_TAG_{prefix_3}_{tag}",
            f"PREFIX_TAG_PREVTAG_{prefix_3}_{tag}_{prev_tag}"
            ]
```

get_feature_dict function is used to generate a dictionary with weights for each feature. These are initialized to 0. **update_weights** is defined to update feature weights globally.

Our main function - **train_perceptron** - takes in data as returned by **data_dump**. We then retrieve a list of all possible states followed by looping through the data splitting them into observations and tags. The **viterbi_perceptron** is used to make predictions using the feature weights defined above and retrieve the states that result in maximum scores using the backward algorithm. These predictions are compared to the gold standard output, where we use the perceptron update rule to penalize wrong predictions. Finally, we have the **get_predictions** function to predict on the development/test dataset.

Through experimentation, we found that a learning rate of 0.1 works the best

for the ES dataset and 0.3 works the best for the RU dataset. For further improvement, we propose adding more features to define a more robust feature vector.

4.3 Results for Part 4

We use the **evalResult.py** again to calculate precision, recall and F score. Our results for part 4 are:

```
#Entity in gold data: 255
#Entity in prediction: 379

#Correct Entity : 165
Entity  precision: 0.4354
Entity  recall: 0.6471
Entity  F: 0.5205

#Correct Sentiment : 130
Sentiment  precision: 0.3430
Sentiment  recall: 0.5098
Sentiment  F: 0.4101
```

Figure 7: Results for ES part 4

```
#Entity in gold data: 461
#Entity in prediction: 314

#Correct Entity : 201
Entity  precision: 0.6401
Entity  recall: 0.4360
Entity  F: 0.5187

#Correct Sentiment : 130
Sentiment  precision: 0.4140
Sentiment  recall: 0.2820
Sentiment  F: 0.3355
```

Figure 8: Results for RU part 4