50.007 Machine Learning Project Report

13 December 2021

Members:
1004422 Harshit Garg
1004472 Krithik Roshan
1004641 Ngui Jia Xuan, Sheriann

Table of Content

Table of Content **2**

1 Part 1 **3**
1.1 Emission Parameters Estimation Using MLE 3
1.2 Emission Parameters Estimation Using MLE with Special Token 3
1.3 Simple Sentiment Analysis System 4
1.4 Evaluation 5

2 Part 2 **7**
2.1 Transition Parameters Estimation Using MLE 7
2.2 Viterbi Algorithm 7
2.3 Evaluation 8

3 Part 3 **10**
3.1 5th Best Output Sequences 10
3.2 Evaluation 10

4 Part 4 **12**
4.1 Second-Order HMM 12
4.2 Improvements 12
4.2.1 Varying K Values 12
4.2.2 Laplace Smoothing 12
4.3 Evaluation 13

5 Evaluation **15**
5.1 ES Dataset 15
5.2 RU Dataset 16
5.3 Test Dataset 16

1 Part 1

All relevant code for part 1 mentioned below is in p1.py.
The code can be run by calling `python p1.py`.

1.1 Emission Parameters Estimation Using MLE

Maximum Likelihood Estimation (MLE) is given by the formula:

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

where x = word token, y = tag

As such, in order to use MLE, we first need to look through the training data provided and store the number of occurrences of each token emitted by the specific tag. The train() function helps us with this by storing emission count in the nested dictionary emission_count, given by the format:

{tag: {token: count, ... }, ...}

When iterating through the lines in the training data, it strips and then splits the data into token and tag. To split the data, rsplit(" ", 1) is used to tackle the special case word tokens ". .." and ". ..." in lines 23167 and 33423 in dataset RU. This python string method splits from the right for a maximum of one time, so that the space inside the word token is not also split and mistakenly taken for its tag. The if-statement after this checks that there are 2 items in the split array, discounting the blank lines, and also acts as a safety net for the case in which either a token or a tag is missing. Once the function has iterated through the training data, emission_count is returned and ready for calculating MLE with.

To calculate MLE, the est_emission_param() function can be called. By passing in the emission_count dictionary returned from the previous function, both numerator and denominator can easily be obtained. For Count(y -> x), emission_count[y][x] is used, and for Count(y), sum(emission_count[y].values()) is used.

1.2 Emission Parameters Estimation Using MLE with Special Token

To consider words that appear in the test set but not the training set, or special tokens, the MLE formula is tweaked slightly to be:

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

To calculate this, we can reuse the train() function to get the number of occurrences of each token emitted by the specific tag, the same as the above.

Then, to calculate MLE with special token, the est_emission_param() function takes in an additional argument k, whose value, for this question, is fixed at 1. Depending on whether the word token appears in the training set, simple if-else branching is used to set the numerator following the formula. Same as the previous part, for Count(y -> x), emission_count[y][x] can be used, and for Count(y), sum(emission_count[y].values()) is used.

1.3 Simple Sentiment Analysis System

The simple sentiment analysis system iterates through both datasets separately, learning the parameters, producing a tag for each word x in the sequence or sentence before writing the token with its tag in an output file.

To learn the parameters, the train() function from above is called to pass on emission_count containing the number of occurrences for evaluation. In this case, we also set the value of k to 1 to deal with the case where words in the test set do not appear in the training set.

Then, to evaluate, a new evaluate() function is called. This function iterates through the test dataset, splitting it into its sentences to get the respective predicted tags by calling another function get_sentence_tag(). The predicted tag should be:

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

for each word x in the sentence

Following the formula, get_sentence_tag() iterates through all word tokens, calculating the emission probabilities with respect to the different tags, and returning only the state with the highest emission probability as the predicted tag.

Finally, these words together with the predicted tags are written into an output file dev.p1.out in the respective data folders.

1.4 Evaluation

The results have been evaluated using the provided evaluation script and calling:

```
python evalResult.py ES/dev.out ES/dev.p1.out
python evalResult.py RU/dev.out RU/dev.p1.out
```

The evaluation results are as follows:

```
ES
#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

==========================
RU
#Entity in gold data: 461
#Entity in prediction: 2089

#Correct Entity : 335
Entity  precision: 0.1604
Entity  recall: 0.7267
Entity  F: 0.2627

#Correct Sentiment : 136
Sentiment  precision: 0.0651
Sentiment  recall: 0.2950
Sentiment  F: 0.1067
```

2 Part 2

All relevant code for part 2 mentioned below is in p2.py.
The code can be run by calling `python p2.py`.

2.1 Transition Parameters Estimation Using MLE

To learn the transition parameter from the training data for first-order Hidden-Markov Model, we need to tabularise the number of occurrences of transiting from a particular state to another particular state.

To do this, we use the transition function, where we go through each line containing a word and its tag, and store the number of occurrences of transitioning from one state to another in a nested dictionary. This transition_count is returned as a nested dictionary in the format:

{ u: { v : count, ...}, ...}

where u and v represent the states when transitioning from one state to another.

We can then use the transitioun_count to get values required to calculate the transition parameters by using the function `transition_param`.

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

The numerator can be obtained with `transition_count[u].get(v,0)` and the denominator by `sum(transition_count[u].values())`. The `transition_param` function uses these values to return the transition parameter.

2.2 Viterbi Algorithm

To get the best path we can use the functions implemented in the previous parts to run the Viterbi Algorithm.

As the natural logarithmic function is monotonically increasing, we calculate our values in natural logarithmic form and use addition instead of division. This is based on the logarithmic property *ln(ab) = ln(a) + ln(b)*. By doing this we reduce the risk of potential numerical underflow. Using natural logarithms, however, introduces a new issue which is that *ln0* = -∞. To handle probability values that are equal to 0, we simply assign ln0 to a large negative integer.

The `viterbi_forward` function keeps track of the score of the best path at each position in a dictionary in the format:

{position: { v: (u, score) } }

where the previous state, u, is stored as a tuple along with the highest score from START to position v, which will later be used for backtracking.

Since the training data is split into sentences as indicated by a single empty line, the base case of $\pi(0, v) = 1$ if v = START, and 0 otherwise is already implicitly accounted for. Our implementation's base case is to determine the score at the position of the first word in a sentence, which is not dependent on the START position.

At each position, we calculate the emission and transition probabilities and get their natural logarithmic values as the score to track. We first check whether the observation is in the training set or not, and if it is not then we assign the special token #UNK#.

Next, we use recursion to go through each sentence. We first initialise the first word of a sentence, such as:

$$\pi(1,v) = \ln[b_v(x_1)]$$

From the second word to the last word of a sentence, we calculate all possible paths and keep track of the state that gives the max score:

$$\pi(k, v) = \max_v \{ \pi(k-1, u) + \ln(a_{u,v}) + \ln[b_v(x_k)]\}$$

Then, we calculate the overall score from START to STOP states. After getting the best path from START to STOP, we do backtracking to retrieve the states at each position. During the recursive process, the state of position k-1 that gave the max score was stored as a tuple along with the score at position k, so we can retrieve the states from the tuple to form the full tag sequence for a sentence.

2.3 Evaluation

The results have been evaluated using the provided evaluation script and calling:

```
python evalResult.py ES/dev.out ES/dev.p2.out
python evalResult.py RU/dev.out RU/dev.p2.out
```

The evaluation results are as follows:

```
ES
#Entity in gold data: 255
#Entity in prediction: 567

#Correct Entity : 131
Entity  precision: 0.2310
Entity  recall: 0.5137
Entity  F: 0.3187

#Correct Sentiment : 104
Sentiment  precision: 0.1834
Sentiment  recall: 0.4078
Sentiment  F: 0.2530

==========================
RU
#Entity in gold data: 461
#Entity in prediction: 535

#Correct Entity : 219
Entity  precision: 0.4093
Entity  recall: 0.4751
Entity  F: 0.4398

#Correct Sentiment : 144
Sentiment  precision: 0.2692
Sentiment  recall: 0.3124
Sentiment  F: 0.2892
```

3 Part 3

All relevant code for part 3 mentioned below is in p3.py.
The code can be run by calling `python p3.py`.

3.1 5th Best Output Sequences

To get the 5th best output sequence, we find the top 5 best paths and store these values at each position. For a node at position k, we obtain all the possible paths from all nodes in position k-1 and calculate the scores of these paths. We then keep track of the scores of the best five paths, along with the path from START up till position k, passing through the node that gives one of the best five scores at position k-1. This process is repeated for all nodes at position k to obtain the best five scores at each node.

Similar to the viterbi algorithm from part 2, the base case of $\pi(0, v) = 1$ if v = START, and 0 otherwise is already implicitly accounted for, as $\ln 1 = 0$. Hence, the base case of our implementation is to determine the score at the position of the first word in a sentence, which is not dependent on the START state position.

At each position, the scores of the best 5 paths for each node are stored in the scores dictionary in the format:

{position: { v : (u1, score), (u2, score), (u3, score), (u4, score), (u5, score)}}

We store the previous state, u, as a tuple with the highest score of the path from START to state v, which will be used later for backtracking.

At each position, we calculate the emission and transition probabilities and get their natural logarithmic values as the score to track. We first check whether the observation is in the training set or not, and if it is not then we assign the special token #UNK#.

For the forward recursive process to calculate the best 5 paths as described above, we first initialise the first word of a sentence, such as:

$\pi(1,v) = \ln[b_v(x_1)]$

From the second word to the last word of a sentence, we calculate all possible paths and keep track of the 5 states that give the top 5 best paths, based on:

$$\pi(k, v) = \max_v \{ \pi(k - 1, u) + \ln(a_{u,v}) + \ln[b_v(x_k)]\}$$

We get the top 5 best scores for each node v and store them and its corresponding preceding node u as a tuple in the scores dictionary as shown above.

Then, we calculate the score of the best 5 paths for each node at position n, where n is the length of a sentence, to the STOP state. After getting all the scores, we filter out the best 5 paths. The scores and its corresponding preceding node are then stored in a tuple.

Then we do backtracking, where we hold N best path sequences in the N_bestPaths list in decreasing order, such as:

[[1st best path], [2nd best path], ... , [Nth best path]]

During the forward recursive process, the top 5 states of position k-1 that gave the max scores were stored in tuples along with the score at position k.

At each current node at position k, we call the scores dictionary at the current position k with the key of the best previous node (at position k-1) for each N best paths. After getting the best previous node to the current node, we add it to the corresponding path in the N_bestPaths list. We do this for all positions until we reach position=1.

Using the N_bestPaths list, we can return the Nth best path.

3.2 Evaluation

The results have been evaluated using the provided evaluation script and calling:

```
python evalResult.py ES/dev.out ES/dev.p3.out
python evalResult.py RU/dev.out RU/dev.p3.out
```

The evaluation results are as follows:

```
ES
#Entity in gold data: 255
#Entity in prediction: 565

#Correct Entity : 119
Entity  precision: 0.2106
Entity  recall: 0.4667
Entity  F: 0.2902

#Correct Sentiment : 92
Sentiment  precision: 0.1628
Sentiment  recall: 0.3608
Sentiment  F: 0.2244

==========================
RU
#Entity in gold data: 461
#Entity in prediction: 444

#Correct Entity : 159
Entity  precision: 0.3581
Entity  recall: 0.3449
Entity  F: 0.3514

#Correct Sentiment : 106
Sentiment  precision: 0.2387
Sentiment  recall: 0.2299
Sentiment  F: 0.2343
```

4 Part 4

All relevant code for part 4 mentioned below is in p4_dev.py.
The code can be run by calling `python p4_dev.py`.

4.1 Second-Order HMM

The second-order Hidden Markov Model works similar to the first-order HMM, with the main difference being that we now consider second-order dependencies in a trigram format. Instead of the prediction of a tag being dependent on just one preceding tag, in second-order HMM it is dependent on the two preceding tags. The order of states that are calculated at each position is defined as u → v → w.

As the natural logarithmic function is monotonically increasing, we calculate our values in natural logarithmic form and use addition instead of division. This is based on the logarithmic property *ln(ab) = ln(a) + ln(b)*. By doing this we reduce the risk of potential numerical underflow. Using natural logarithms, however, introduces a new issue which is that *ln0* = -∞. To handle probability values that are equal to 0, we simply assign ln0 to a large negative integer.

Similar to the viterbi algorithms implemented in the previous parts above, the base case of π(0, START, START) = 1 if v, w = START, and 0 otherwise is already implicitly accounted for, as ln1 = 0. Our implementation's base case is to determine the score at the position of the first word in a sentence.

At each position, the score of the best path is stored in a dictionary, scores, in the format:

{ w: { u: { v: score } } }

We store the highest score of the previous 2 states, u and v, as the value of the nested dictionary from START to state w, which will be used later for backtracking.

At each position, we calculate the emission and transition probabilities and get their natural logarithmic values as the score to track. We first check whether the observation is in the training set or not, and if it is not then we assign the special token #UNK#.

Our function now returns transition_count as a nested dictionary in the format:

{ u: { v: { w: count_u_v_w } } }

We can use this to retrieve the values required to calculate the transition parameters, using the transition_param function.

For the forward recursive process, we first initialise the first word of a sentence, such as:

$\pi(1,v) = \ln[b_v(x_1)]$

Next, we have two cases:

1. 1 word sentence and 2 word sentences
    a. In both cases, we calculate the possible paths and keep track of the state that gives the mac score. We need to look up the scores dictionary for the correct previous states π(*k* − 1, *u*, *v*).
        i. $\pi(k, v, w) = \max_v \{ \pi(k - 1, u, v) + \ln(a_{u,v,w}) + \ln[b_w(x_k)]\}$

2. More than 2 word sentences
    a. From the second word to the last word of a sentence, we calculate all possible paths and keep track of the state that gives the maximum score:
        i. $\pi(k, v, w) = \max_v \{ \pi(k - 1, u, v) + \ln(a_{u,v,w}) + \ln[b_w(x_k)]\}$

In the final step, we calculate the score of the overall path from START to STOP. Then, we do backtracking to retrieve the states at each position. During the recursive process, the state of position k-1 that gave the max score was stored as a tuple along with the score at position k, so we can retrieve the states from the tuple to form the full tag sequence for a sentence.

4.2 Improvements

To improve on our sentiment analysis system, we tried the following modifications:

1. Varying k values (*Section 4.2.1*)
2. Laplace smoothing (*Section 4.2.2*)

4.2.1 Varying K Values

To find the k value that optimises test accuracy, we tested k values for $k = 10^x$, where $-1 < x < 9$ is a random value. We chose the k value with the highest accuracy, and based on our tests it's likely to occur when k=3. Although this change in F score was quite small but but this change still might overfit the data.

4.2.2 Laplace Smoothing

As our dataset is limited, we thought of trying smoothing methods that work by assigning a non-zero probability to unseen words, improving the accuracy of word probability estimation in general.

For Laplace smoothing, also known as add-one smoothing, 1 is added to all the counts before calculating probability. The equation is as follows:

$$e(x, y) = \frac{Count(y \to x) + 1}{\sum_x [Count(y) + 1]}$$

After implementing it, we noticed that our accuracy dropped from 0.3308 to 0.3123. As such, it was removed from our final sentiment analysis system.

Given more time to conduct proper cross-evaluation, we would tweak the constant added to the count, following the equation:

$$e(x, y) = \frac{Count(y \to x) + \alpha}{N + \alpha d}$$

where α= constant to vary, d = number of classes

For α, a value striking a balance in the bias-variance tradeoff should be used. If α is too small, it will lead to high variance, and if it is too large, it will lead to high bias.

4.3 Evaluation

The results have been evaluated using the provided evaluation script and calling:

```
python evalResult.py ES/dev.out ES/dev.p4.out
python evalResult.py RU/dev.out RU/dev.p4.out
```

The evaluation results are as follows:

```
ES
```

```
#Entity in gold data: 255
#Entity in prediction: 404

#Correct Entity : 137
Entity  precision: 0.3391
Entity  recall: 0.5373
Entity  F: 0.4158

#Correct Sentiment : 109
Sentiment  precision: 0.2698
Sentiment  recall: 0.4275
Sentiment  F: 0.3308
==========================
RU
#Entity in gold data: 461
#Entity in prediction: 737

#Correct Entity : 239
Entity  precision: 0.3243
Entity  recall: 0.5184
Entity  F: 0.3990

#Correct Sentiment : 155
Sentiment  precision: 0.2103
Sentiment  recall: 0.3362
Sentiment  F: 0.2588
```

5 Evaluation

The scores for both ES and RU datasets across part 1 to 4 have been plotted in *Figure 5.1a* and *5.2a* respectively.

5.1 ES Dataset

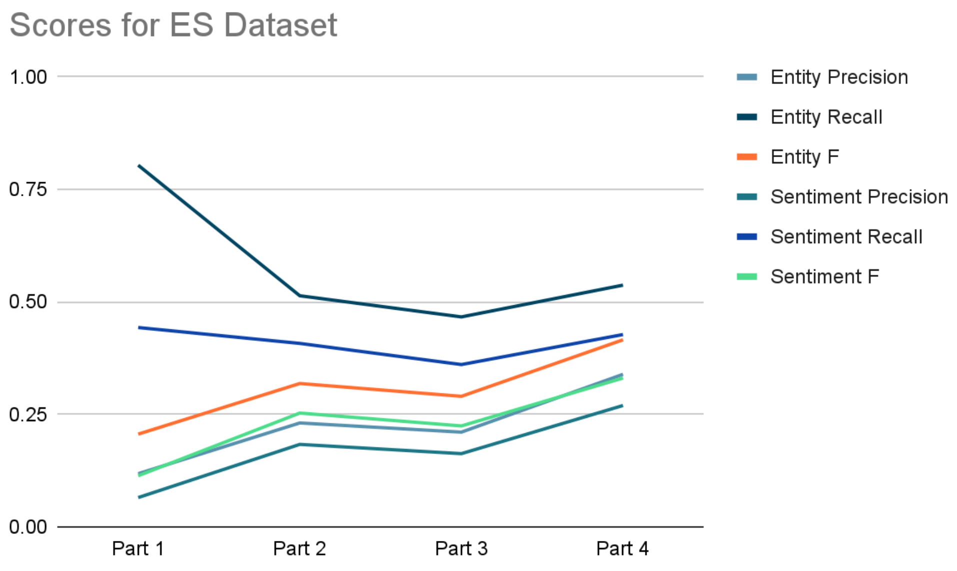

Figure 5.1a: Scores for ES Dataset Across Parts 1 to 4

For the ES dataset, we observed a general increase in scores, as expected.

5.2 RU Dataset

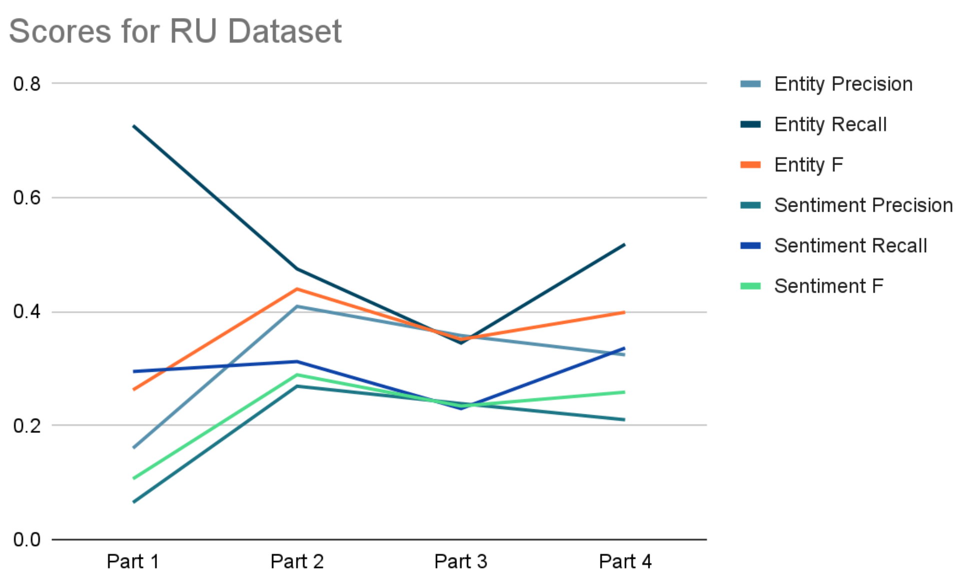

Figure 5.2a: Scores for RU Dataset Across Parts 1 to 4

For the RU Dataset, we observed that the scores are more mixed. However, for entity and sentiment precision, both have a decreasing trend despite using more complicated models.

Upon further inspection of the dataset, we noticed that the RU dataset has significantly more punctuations which may be decreasing the scores. Due to the time constraint, we left it as it is, but given more time, we would have conducted data cleaning on the dataset before using the sentiment analysis system. We would try removing all, or if not just successive duplicated punctuations, and evaluating from there. Ideally, the scores would increase as there is less ambiguity caused by the punctuations.

5.3 Test Dataset

Since the test dataset is to be tested on the sentiment analysis system from part 4, we simply make use of the same file but add an additional '-test' argument when running. The command to use is: `python p4_dev.py -test`.

The script will use the dev datasets for training, before evaluating on the test datasets and writing an output to test.p4.out in the respective test dataset folders.