MACHINE LEARNING DESIGN PROJECT

50.007 MACHINE LEARNING

ELLIOT KOH 1003501

SEAN LIM 1003377

SIDHARTH PRAVEEN 1003647

INFORMATION SYSTEMS AND TECHNOLOGY DESIGN

DECEMBER, 2020

Contents

Chapter 1

Part 2: Estimation of Emission Parameters

1.1 Approach and Motivation

1.1.1 Emission Parameters

The estimation of emission parameters $b_i(o)$ for a Hidden Markov Model(HMM) can be calculated by the simple formula:

$$b_i(o) = \frac{count(i \rightarrow o)}{count(i)} \tag{1.1}$$

This equation is formulated from the Maximum Likelihood Estimation where $count(i -> o)$ is the number of times a sequence o was emitted from a given label(tag) i. Dividing this with the total number of times i appears in the training set gives us the emission probability.

1.1.2 Fixed Emission Parameters

To account for words that appear in the test set but not in the train set, we use a UNK token to signify them as unknown sequences. Equation 1 is no longer valid, so we make use of a Laplace Smoothing factor, $k = 0.5$ with the following emission formula:

$$b_i(o) = \frac{k}{count(i) + k} \tag{1.2}$$

1.1.3 Functions

In *part2.py* of the submission, the main function is `getEmissions(file, k)` which returns a hash-map of the form `{i: {o : b}}` where i and o are the sequence and tags respectively, while b is the corresponding emission probability.

In order to save on computational time, we use a hash-map to store all the emission pairs and probabilites because of it's constant look-up advantage. This enables us to save on a lot of computational time by searching using the key as opposed to traditional array-like approaches.

The function parses each line of the training input into the sequence and the tag (x and y in the function), and iteratively increments the frequencies for each pair of i-o combinations.In the end, we iterate through the hash-map once more to divide all the frequencies by the total observation counts of that specific label to get the corresponding emission probabilities.

1.2 Algorithm and Model

For a given test set, to predict the tag for a given sequence by looking up the hash-map to select the tag with the highest emission probability. This is done with the help of `predictSentiments(emissions,testfile, , outputfile)` which writes the predictions to the file *dev.p2.out* for the given test set *dev.in*

1.3 Results

We perform the evaluation by running *evalRun.py* which was given to us, and observe that the sequence labelling with emission parameters gave the best F scores for the EN Dataset, 0.5244, followed by a F score of 0.1783 for the SG Dataset and 0.0680 for the CN Dataset.

```
EN                                  SG
#Entity in gold data: 13179         #Entity in gold data: 4301
#Entity in prediction: 18847        #Entity in prediction: 12415

#Correct Entity : 9489              #Correct Entity : 2381
Entity  precision: 0.5035           Entity  precision: 0.1918
Entity  recall: 0.7200              Entity  recall: 0.5536
Entity  F: 0.5926                   Entity  F: 0.2849

#Correct Sentiment : 8397           #Correct Sentiment : 1490
Sentiment  precision: 0.4455        Sentiment  precision: 0.1200
Sentiment  recall: 0.6372           Sentiment  recall: 0.3464
Sentiment  F: 0.5244                Sentiment  F: 0.1783

                    CN
                    #Entity in gold data: 700
                    #Entity in prediction: 4244

                    #Correct Entity : 345
                    Entity  precision: 0.0813
                    Entity  recall: 0.4929
                    Entity  F: 0.1396

                    #Correct Sentiment : 168
                    Sentiment  precision: 0.0396
                    Sentiment  recall: 0.2400
                    Sentiment  F: 0.0680
```

Figure 1.1: Fig 1. Evaluation results for Part 2

Chapter 2

Part 3: Estimation of Transition Parameters and Prediction using Viterbi Algorithm

2.1 Approach and Motivation

We observe that relying on the emission parameters solely lead to unsatisfactory accuracies and F1 scores. This is because by restricting our focus on each input tag i to generate a corresponding observation o, we are making an assumption that context is irrelevant and the words before or after the input tag do not make a difference in the output. This assumption however, cannot be applied to real life scenarios, and so the model performs poorly on the test set. To further enhance the model, we take into account the transition parameters, and optimize it using the Viterbi Algorithm.

2.1.1 Estimating Transition Parameters

The estimation of transition parameters a_{ij} for a Hidden Markov Model(HMM) can be calculated by the formula:

$$a_i, j = \frac{count(i, j)}{count(i)} \tag{2.1}$$

This equation is formulated from the Maximum Likelihood Estimation where $count(i, j)$ is the number of times there existed a transition from i to j in the training set, where i and j are tags. Dividing this with the total number of times i appears in the training set gives us the transition probability.

2.1.2 Transition Functions

In *part*3.*py* of the submission, the main function is `getTransitions(file)` which returns a hash-map of the form `{i: {j : a}}` where i and j are the tags, while a is the corresponding transition probability.

In order to save on computational time, we use a hash-map to store all the emission pairs and probabilities because of it's constant look-up advantage. This enables us to save on a lot of computational time by searching using the key as opposed to traditional array-like approaches.

The function parses each line of the training input into the sequence and the tag (x and y in the function), and iteratively increments the frequencies for each pair of i-o combinations.In the end, we iterate through the hash-map once more to divide all the frequencies by the total observation counts of that specific label to get the corresponding emission probabilities.

2.2 Algorithm and Model

While following HMM, the joint probability of a given n-length sequence, with a and b as transition and emission parameters respectively, is given as:

$$P(x_1, x_2, ...x_n, y_1, y_2, y_n) = \prod_{i=1}^{n+1} a_{y_{i-1},i} \prod_{i=1}^{n} b_{y_i}(x_i) \tag{2.2}$$

Given a set of T possible tags, calculating the joint probability for a given sequence of length n will be exponential in terms of asymptotic complexity. The asymptotic running time of a naive implementation for such a model would be $O(T^n)$. To optimize this algorithm, we make use of the Viterbi Algorithm, which uses a dynamic programming approach to decrease the asymptotic complexity to $O(nT^2)$.

Though this is the theoretical approach for the implementation of Viterbi Algorithm, we have made a slight modification to combat underflow problems, arising from the multiplication of probabilities ¡ 0. So we adopted the log joint probability, which gives:

$$logP(x_1, x_2, ...x_n, y_1, y_2, y_n) = \sum_{i=1}^{n+1} log(a_{y_{i-1},i}) + \sum_{i=1}^{n} log(b_{y_i}(x_i)) \tag{2.3}$$

For the implementation of Viterbi Algorithm, we define a hashmap *score* in function $ViterbiAlgo()$ of the format `{State number : {Tag: {Highest Score: Parent Tag}}`. We populate this hashmap by iterating through each line of the test set, and checking if that word was present in the training set. If it wasn't, we replace it with the UNK tag. Still inside the former loop, we iterate through each tag and set its score and parent as None initially. We then check the previous elements of score iteratively and compute the score using eqn 2.3. If this tag is present in score with a lower score than the current score, then we update the value, else we continue. This process goes on for the entire test set, after which we append the STOP state and the score hashmap will be fully populated, with each tag being associated with its 'best' parent. To backtrack, we iterate from the end state to the start state of the hashmap. We pick the parent that has the highest score for that specific tag, and then append the sequence associated to the complete output.

Algorithm 1: Viterbi Algorithm (Forward Pass)

```
Result: Write here the result
score = START, [0, None];
for each line in test set do
    word = first index of line;
    if word not in train set then
        word = UNK;
    else
        for each tag in emissions do
            let b = emission probability ;
            for previousTag in score do
                let a = transition probability;
                score = log(a) + log(b) + prevScore;
                update score[i] if score > score[i][tag]
            end
        end
    end
end
```

Algorithm 2: Viterbi Algorithm (Backward Pass)

```
Result: Write here the result
score = START, [0, None];
for each line in test set do
    word = first index of line;
    if word not in train set then
        word = UNK;
    else
        for each tag in emissions do
            let b = emission probability ;
            for previousTag in score do
                let a = transition probability;
                score = log(a) + log(b) + prevScore;
                update score[i] if score > score[i][tag]
            end
        end
    end
end
```

2.3 Results

We perform the evaluation by running *evalRun.py* which was given to us, and observe that the sequence labelling with emission parameters gave the best F scores for the EN Dataset, 0.7323, followed by a F score of 0.4210 for the SG Dataset and 0.1588 for the CN Dataset.

```
EN                                  SG
#Entity in gold data: 13179         #Entity in gold data: 4301
#Entity in prediction: 14006        #Entity in prediction: 4250

#Correct Entity : 10585             #Correct Entity : 2177
Entity  precision: 0.7557           Entity  precision: 0.5122
Entity  recall: 0.8032              Entity  recall: 0.5062
Entity  F: 0.7787                   Entity  F: 0.5092

#Correct Sentiment : 9954           #Correct Sentiment : 1800
Sentiment  precision: 0.7107        Sentiment  precision: 0.4235
Sentiment  recall: 0.7553           Sentiment  recall: 0.4185
Sentiment  F: 0.7323                Sentiment  F: 0.4210

                    CN
                    #Entity in gold data: 700
                    #Entity in prediction: 849

                    #Correct Entity : 206
                    Entity  precision: 0.2426
                    Entity  recall: 0.2943
                    Entity  F: 0.2660

                    #Correct Sentiment : 123
                    Sentiment  precision: 0.1449
                    Sentiment  recall: 0.1757
                    Sentiment  F: 0.1588
```

Figure 2.1: Fig 1. Evaluation results for Part 3

Chapter 3

Part 4: Prediction of 3rd Best Output Sequence

3.1 Approach and Motivation

Following the implementation of first order Viterbi algorithm, the next task was to modify the approach to find the third best output sequence. Initially, the idea was to simply have the same forward algorithm as the shown in Part 3 followed by taking the third highest value at each step while backtracking. However, we realized that picking the third best sequence iteratively would not lead to extraction of the third best output sequence in the global context (of train set), and so we had to shift our approach.

The approach was to amend both the forward and backward portion of the Viterbi algorithm. Moving forward, instead of storing the largest score per tag, each tag would now consist of the three highest scores obtained. Moving backward (to obtain the tag sequence), the third highest score at the stop tag would start as the starting point. After which, the full path is determined by deducting transmission and emission scores to see which of the three parent node did this current tag come from.

3.2 Algorithm and Model

For the implementation of the amended Kth-Best Virterbi algorithm, we define a hashmap score of the format:

```
{State number : {Tag: [[Kth Highest Score, Associated Parent Tag]
,[K-1th Highest Score, Associated Parent Tag]...[Highest Score, Associated Parent Tag ]}.
```

Hence for the third best (K = 3) the hashmap score would be of the format:

```
{State number : {Tag: [[Third Highest Score, Associated Parent Tag],
[Second Highest Score, Associated Parent Tag],[Highest Score, Associated Parent Tag ]}
```

Moving forward, we populate this hashmap by iterating through each line of the test set, and checking if that word was present in the in the training set. Similar to the algorithm above, if it was not, the UNK tag would replace that word. Still inside the former loop, we iterate through each tag and set its each of the three highest score and associated parent as `[None,None]` initially. We then check the previous elements of each of the 3 scores iteratively and compute the scores using eqn2.3. This is done until the top 3 scores and their associated parent tag are updated in the hashmap. This process goes on for the entire test set, after which we append the STOP state and the score hashmap will be fully populated.

To backtrack, the model could be split up into 3 parts. (1) Initialisation (2) nth tag handling (3) Continuation to "START" tag

(1) **Initialisation**
Assuming K=3, an empty prediction array is initialised. At the 'STOP' tag, third highest score and it's associated parent tag would each be a stored in a separate variable. The parent tag would be appended into the prediction array. After which, a `POST` variable will be assigned to 'STOP', a `CURR` variable will be assigned to the chosen parent tag and a `K_BEST_SCORE` variable will be assigned to the third highest score. For example, if the hashmap was something like `{n : {"STOP": [[2, "O"],[ 5, "I"],[6, "B"]}`, the `K_BEST_SCORE` would equal to 2, `CURR` would equal to "O", `POST` would equal to "STOP" and the prediction array would equal to ["O"] at this stage.

(2) **nth tag handling**
As from the nth tag to 'STOP' tag, there's a transition value but no emission value. Hence, only for this sequence, we would determine the nth score by seeing which of it's three values, upon adding log(transition value) would equate to `K_BEST_SCORE`. Upon finding the value that does, the associated parent tag would be appended into the prediction array, `CURR` would be assigned to the associated parent tag of the value, `POST` would be assigned to the nth tag and `K_BEST_SCORE` would be assigned to the chosen value. For example, if the hashmap was `{n-1 : {"O": [[1, "O"],[ 1.5, "I"],[1.7, "B"]}`, and log(transition value) equals to 0.5, the new chosen `K_BEST_SCORE` would equal to 1.5, `CURR` would equal to "I", `POST` would equal to "O" and the prediction array would equal to ["O","I"] at this stage.

(3) **Continuation to "START" tag**
From here on until the 'START' tag is reached, we must now consider both the emission value of each word for the `POST` tag and the transition value of the `CURR` tag to the `POST` tag. We would then determine each ith by seeing which of it's three values, upon adding log(transition value) and log(emission value) would equate to `K_BEST_SCORE`. Upon finding the value that does, the associated parent tag would be appended into the prediction array, `CURR` would be assigned to the associated parent tag of the value, `POST` would be assigned to the nth tag and `K_BEST_SCORE` would be assigned to the chosen value. This would be done until the 'START' is reached and finally, the prediction array would be reversed in order to see the arranged predicted sequence of tags given a sentence.

For example, if the hashmap given was `{1 (n-2) : {"I": [[1.25, "START"],[ None, None],[None, None]}` and `{0 : {"START": [[0.0, None],[ 0.0, None],[0.0, None]}`. At the step to choose the score for the previously chosen "I" tag. Lets say the log(transition value from "I" to "O") equals to 0.2, and the log(emission value of word[2] at "O") is 0.05 the new chosen `K_BEST_SCORE` would equal to 1.25, `CURR` would equal to "START", `POST` would equal to "I". However, since "START" is reached, the iterative loop will end the prediction array would be maintained at ["O","I"]. This would however be reversed at the end making the final prediction tags to be ["I","O"].

Algorithm 3: 3rd Best Viterbi Algorithm (Forward Pass)

```
score = {0 : {START : [[0, None], [0, None], [0, None]]}} //Initialisation;
for each line in test set do
    word = first index of line;
    if word not in train set then
        word = UNK;
    else
        for each tag in emissions do
            let highScores = [[None,None],[None,None],[None,None]];
            let b = emission probability[tag][word] ;
            for previousTag, previousScores in score do
                let a = transition probability[previousTag][tag];
                for each previousScore of previousScores do
                    let highScore = previousScore + log(a) + log(b) for p in range(3) do
                        if highScores[p][0] is None then
                            highScores[p][0] = highScore;
                            highScores[p][1] = previousTag;
                            break out of loop;
                        else
                            if highScore == highScores[p-1][0] then
                                break out of loop //prevent duplicates
                            else
                                if highScore > highScores[p][0] then
                                    highScores[p][0] = highScore;
                                    highScores[p][1] = previousTag;
                                    break out of loop;
                                else
                                    continue;
                                end
                            end
                        end
                    end
                end
            end
            score[index of word in line]= {tag: highScores};
        end
        // Stopping Case let highScores = [[None,None],[None,None],[None,None]];
        for previousTag, previousScores in score do
            let a = transition probability[previousTag]["STOP"];
            for each previousScore of previousScores do
                let highScore = previousScore + log(a);
                Take top 3 highScore from all previousScore and store it into highScores;
            end
        end
        score[line length + 1] = {"STOP": highScores };
    end
end
```

Algorithm 4: 3rd Best Viterbi Algorithm (Backward Pass)

```
//(Initialisation Step);
Get finalScore from forward algorithm;
let prediction = [];
let CURR = "STOP";
let POST = "";
let i = length of textList;
let kScoresAtNode = finalScore[i+1][CURR];
let K − BEST − SCORE = 10000 //arbitrary max value;
for each score in kScoresAtNode do
    K − BEST − SCORE = Third Highest Score;
    if All Scores == None then
        let parent = None;
        K − BEST − SCORE = None;
    else
end
let index = index of K − BEST − SCORE in kScoresAtNode;
parent = finalScore[i+1][CURR][index][1];
prediction.append(parent);
POST = CURR; CURR = parent; i -= 1;
//(nth tag handling Step);
let kScoresAtNode = finalScore[i+1][CURR];
let a = transition probability[CURR][POST] for each score in kScoresAtNode do
    if check for score[0] + log(a) == K − BEST − SCORE then
        let parent = score[1];
        K − BEST − SCORE = score[0];
    else
    end
end
prediction.append(parent);
POST = CURR; CURR = parent; i -= 1;
//(continuation to "START" tag Step);
while True do
    let word = textList[i+1] if word not in train set then
        word = UNK;
    else
    end
    let kScoresAtNode = finalScore[i+1][CURR];
    a = transition probability[CURR][POST];
    b = emission probability[POST][WORD];
    for each score in kScoresAtNode do
        if check for score[0] + log(a) + log(b) == K − BEST − SCORE then
            let parent = score[1];
            K − BEST − SCORE = score[0];
        else
    end
    if parent == "START" then
        break  end while true loop
    else
    end
    prediction.append(parent);
    POST = CURR; CURR = parent; i -= 1;
end
RETURN prediction.reverse()
```

3.3 Results

```
EN
#Entity in gold data: 13179
#Entity in prediction: 14192

#Correct Entity : 9868
Entity  precision: 0.6953
Entity  recall: 0.7488
Entity  F: 0.7411

#Correct Sentiment : 9077
Sentiment  precision: 0.7118
Sentiment  recall: 0.7180
Sentiment  F: 0.7010
```

Figure 3.1: Fig 1. Evaluation results for Part 4

Though there is no output set for the third best output sequence, we evaluate the model by training it with EN *train* set and then comparing the generated tags with *dev.out*. We can follow intuition and assert that the model does the job, owing to the dip in F scores for entities and sentiment. This is trivial because comparison with outputs from Part 3 consist of the 'best' output sequence.

Chapter 4

Part 5: Design Challenge

4.1 Approach and Motivation

To enhance the performance of our first order Hidden Markov Model, we explored a few discriminative approaches instead of generative. The difference between generative and discriminative approaches in the context of Sequence Labelling / POS Tagging is that in generative we generate the word sequences given the tag, but in discriminative, it becomes like that of a classifier and predicts the tag given words.

4.1.1 Maximum Entropy Markov Model

The Maximum Entropy Markov Model, also known as the Conditional Markov Model is a widely used sequence labelling algorithm that mixes the theories behind Hidden Markov Models and Maximum Entropy Models. In simple words, it is a Hidden Markov Model that now leverages on Backward Emissions c and Forward Emissions$d$ (from the words before and after a given word). The emission probability now, however, is reversed, i.e, emission now refers to a word emitting a tag. The conditional probability for emission and transition in this model, with the weights w, given words x and tags y is:

$$P(y_1, y_2, ...y_n, x_1, x_2, x_n) = \prod_{i=1}^{n+1} w_1 a_{y_{i-1},i} \prod_{i=1}^{n} w_2 b_{xi}(y_i) \prod_{i=1}^{n-1} w_3 c_{xi}(y_i) \prod_{i=1}^{n+1} w_4 d_{xi}(y_i) \tag{4.1}$$

4.2 Algorithm and Model

The algorithm runs similar to that of part 3, with the inclusion of additional emission lists now. The backward emission c and forward emissions d contain dictionaries of the format `{word:{tag:b}`. The transition probabilities remain the same as that of the HMM. For sequence label prediction, we implemented the function `discriminativeViterbiAlgo()` that runs just like the Viterbi Algorithm in Part 3, with an updated scoring function that takes into account the backward and forward emission probabilities as well. We decided to add weights as hyperparameters to these emissions. To combat underflow, like in Part 3, we do calculate the log sum instead of multiplying the probabilites together. The score for each node, is thus given as:

$$logP(y_1, y_2, ...y_n, x_1, x_2, x_n) = \sum_{i=1}^{n+1} w_1 log(a_{y_{i-1},i}) + \sum_{i=1}^{n} w_2 log(b_{xi}(y_i)) + \sum_{i=1}^{n-1} w_3 log(b_{xi}(y_i)) + \sum_{i=1}^{n+1} w_4 log(b_{xi}(y_i)) \tag{4.2}$$

To obtain the optimal weights, we performed fine tuning of hyperparameter by iterating it through a set of values and recorded them as follows:

```
w_1 = 3.3, w_2 = 6, w_3 = 1.5, w_4 = 0.1
```

4.3 Results

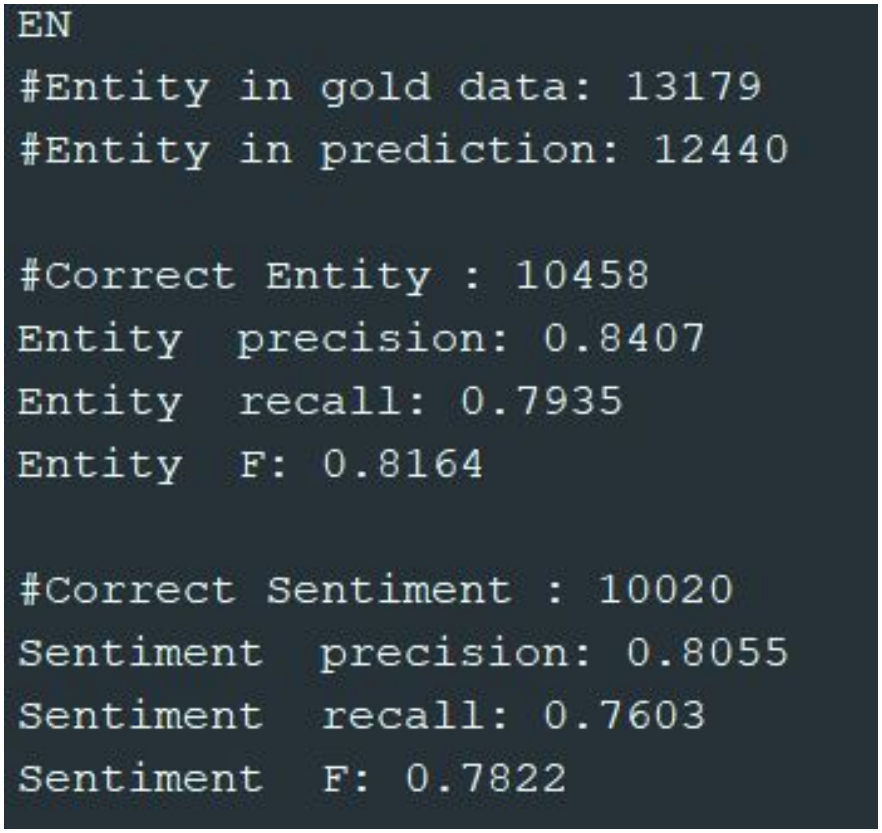

Figure 4.1: Fig 1. Evaluation results for Part 5

As can be observed, our MEMM model performed better than the first order Viterbi Algorithm, giving us a sentiment F score of 0.7822.