50.007 MACHINE LEARNING

DESIGN PROJECT

SENTIMENT ANALYSIS SYSTEM

Authors

Vikramaditya Roy [1005414] vikramaditya_roy@mymail.sutd.edu.sg

Saakshi Vinod Saraf [1005380] saakshi_saraf@mymail.sutd.edu.sg

Contents

Part 1

1.1 Approach:

The provided code is a Python script that predicts the sequence labels of words in a text file using a pre-trained model based on the emission probabilities. The approach used in this script is the Hidden Markov Model (HMM) which is a statistical model that uses probability distributions to predict a sequence of hidden states. The motivation behind this script is to classify words into their respective tags based on the training data and pre-trained model.

To estimate the emission parameters bi(o) in a Hidden Markov Model (HMM), we can use the following straightforward formula:

$$bi(o) = \frac{count(i \rightarrow o)}{count(i)}$$

This equation is derived from the Maximum Likelihood Estimation approach, where count(i → o) represents the number of times a particular label tag 'i' emits the sequence o. Dividing this count by the total number of times label i appears in the training set yields the emission probability.

In order to handle words that are present in the test set but absent in the training set, we introduce a special UNK token to represent these unknown sequences. Since Equation 1 becomes inappropriate in this scenario, we apply Laplace Smoothing with a smoothing factor of k = 0.5. This enables us to utilize the following emission formula instead:

$$bi(o) = \frac{k}{count(i) + k}$$

1.2 Functions:

`Counts(parent, child, d)`:

This function takes three parameters, parent, child, and d, and defines and increments the count of [parent][child] in a dictionary d. The purpose of this function is to count the occurrences of each tag (i) and word in the training data.

`Emission(file, k=0.5)`:

This function takes two parameters, a file (dev.in) from each of the 2 files EN or FR and a value k (default 0.5). The purpose of this function is to generate the emission parameters (dict) used in predicting the sequence labels of words. It replaces words that appear less than k times with #UNK# and stores the dictionary format as {i: {o:emission prob}}.

`PredictionSentiments(emission, testfile, outputfile)`:

This function takes three parameters, emission (emission parameters generated by the Emission function), testfile (the file to be classified), and outputfile (the file where the predicted sequence labels are to be stored). The purpose of this function is to predict the sequence labels using argmax(emission) and find the best #UNK# for later use.

`main(args)`:

This function takes one parameter, args (either EN or FR). The purpose of this function is to input the necessary files and define the output file dev.p1.out.

1.3 Algorithm and Model:

The algorithm used in this script is the Hidden Markov Model (HMM) which is a statistical model that uses probability distributions to predict a sequence of hidden states. The HMM model assumes that there is an underlying hidden state that generates the observed data. In this script, the HMM model is used to predict the sequence labels of words in the text file.The HMM model is trained on a set of training data to generate the emission parameters. The emission parameters are used to predict the sequence labels of words in the test file. The emission parameters are generated by the `Emission():` function, which replaces words that appear less than k times with #UNK# and stores the dictionary format as {i: {o:emission prob}}.

The predicted sequence labels of words are generated by the `PredictionSentiments():` function, which uses emission parameters to predict the sequence labels of words in the test file using `argmax(emission):` and finds the best #UNK# variable for later use.

The `main():` function inputs the necessary files and defines the output file dev.p1.out. It then calls the `Emission():` and `PredictionSentiments():` functions to generate the predicted sequence labels of words in the test file. The results of this implementation is as follows:

1.4 Results

```
#Entity in gold data: 802
#Entity in prediction: 1040

#Correct Entity : 559
Entity  precision: 0.5375
Entity  recall: 0.6970
Entity  F: 0.6069

#Correct Sentiment : 450
Sentiment  precision: 0.4327
Sentiment  recall: 0.5611
Sentiment  F: 0.4886
```

Prediction for EN

```
#Entity in gold data: 238
#Entity in prediction: 1016

#Correct Entity : 192
Entity  precision: 0.1890
Entity  recall: 0.8067
Entity  F: 0.3062

#Correct Sentiment : 88
Sentiment  precision: 0.0866
Sentiment  recall: 0.3697
Sentiment  F: 0.1404
```

Prediction for FR

Part 2

2.1 Approach:

The Viterbi algorithm is a dynamic programming algorithm used to find the most likely sequence of hidden states given a sequence of observations. In the context of HMM, the hidden states represent the state of the system, which is not directly observable, and the observations are the outputs of the system.
Using the MLE (maximum likelihood estimation):

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

This function can estimate the transition probability.

2.2 Functions Used:

`Transition(file):`
This function reads a training file and returns the transition probabilities of the HMM in the form of a dictionary. It also returns a dictionary of counts of occurrences of each state in the training data.

`UniqVocab(file):`
This function reads a training file and returns a set of unique words present in the file.

`Missing(child, parent, hashmap):`
This function is used to check whether a child's parent is actually the parent in the given dictionary.

`Viterbi(emission, transition, vocab, lines):`
This function takes as input the emission probabilities, transition probabilities, the vocabulary of the training data, and the test data in the form of a list of sentences. It implements the Viterbi algorithm to find the most likely sequence of hidden states for each sentence.

`ViterbiPrediction(emission, transition, vocab, inputFile, outputFile):`

This function is used to read the test file, call the Viterbi function, and write the predictions to an output file.

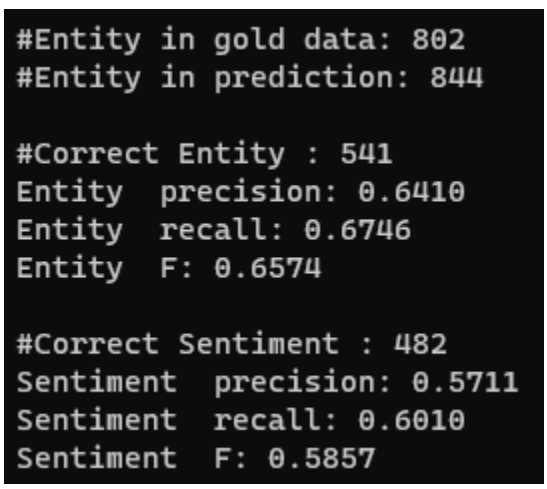
2.3 Algorithm and Model:

The Viterbi algorithm is a dynamic programming (DP) algorithm that computes the most likely sequence of hidden states when we are given a sequence of observations. This algorithm works by maintaining a table of scores for each possible state at each time step. Each cell in the table or in the file in this case represents the probability of being in that state at that time step given the observations up to that point.

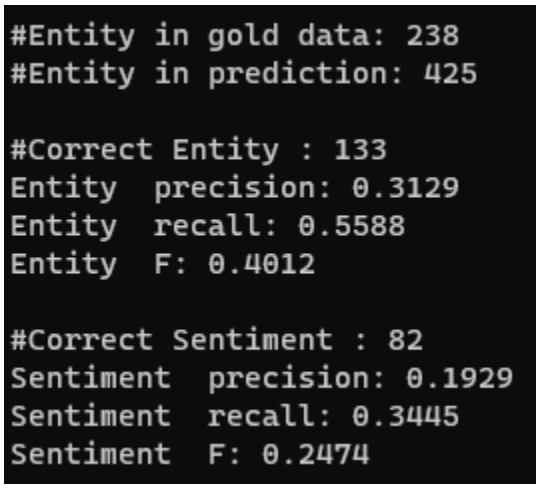
The HMM model used in part 1 consists of a set of states, a set of observations, and two sets of probabilities: the transition probabilities and the emission probabilities. The transition probabilities represent the probability of moving from one state to another, and the emission probabilities instead, give the probability of emitting an observation from each state. The HMM assumes that the probability of emitting an observation only depends on the state that emitted it, and the probability of transitioning to a new state, which depends only on the current state.

This code estimates the transition and emission probabilities from the training data and uses them to predict the most likely sequence of hidden states for each sentence in the test data using the Viterbi algorithm. These predictions are then written to an output file called dev.p2.out. The results are as follows:

2.4 Results



Prediction for EN



Prediction for FR

Part 3

3.1 Approach:

The Named Entity Recognition (NER) model uses the Viterbi algorithm just as in part 2 to predict the tags for the words in a given text. It is a statistical approach that uses training data to learn the patterns and relationships between words and their corresponding tags.

A visual representation of NER

The model makes use of Hidden Markov Models (HMMs) to predict the most likely sequence of tags for a given input sentence. The model takes as input a dataset containing sentences and their corresponding tags, and outputs a predicted tag sequence for each sentence. However unlike the previous part, the model should be able to take second-order dependencies with the HMM being parametrized as follows:

$$p(x_1, \ldots, x_n, y_{-1}, y_0, y_1, y_2, \ldots, y_n, y_{n+1}) = \prod_{i=1}^{n+1} q(y_i|y_{i-2}, y_{i-1}) \cdot \prod_{i=1}^{n} e(x_i|y_i)$$

As far as the time complexity of the algorithm goes:
The time complexity of the `Emission():` function is O(N), where N is the number of words in the training corpus. The time complexity of the `Transition():` function is O(N^2),

where N is the number of tags in the training corpus. The time complexity of the `UniqVocab():` function is O(N), where N is the number of words in the training files.

The time complexity of the `Viterbi():` function is O(T*N^2), where T is the length of the sentence and N is the number of tags in the training files.

The time complexity of the `ViterbiPrediction():` function depends on the length of the sentences in the input file. For each sentence, the Viterbi() function is called, which has a time complexity of O(TN^2). Therefore, the time complexity of the `ViterbiPrediction() :` function is O(ST*N^2), where S is the number of sentences in the input file, T is the length of the longest sentence in the input file, and N is the number of tags in the training files. This function dominates the overall time complexity of the algorithm.

3.2 Functions Used:

`Emission():`

This function calculates the emission probabilities for each word and tag pair in the training dataset.

`Counts(parent1, parent2, child, d):`

Unlike the previous Counts function, this function now takes four parameters, parent1, parent2, child, and d, and defines and increments the count of [parent1][parent2][child] in a nested dictionary d.

`Transition():`

This function calculates the transition probabilities for each tag pair in the training dataset. The difference from the earlier transition function is that this function returns a dictionary of the format :

{ yi-2 : {yi-1: {yi : probability of transition from yi-2 to yi-1 to yi}}}

Thus it calculates second-order probabilities for the modified version unlike the original, which had a probability of transition from yi-1 to yi

`UniqVocab():`

This function creates a set of unique words so as to process them in the training dataset.

`Viterbi():`
This function implements the Viterbi algorithm with the updated parameters to predict the most likely sequence of tags for a given sentence.

`ViterbiPrediction():`
This function uses the aforementioned Viterbi algorithm to thus predict the tag sequence for each sentence in the given datasets.

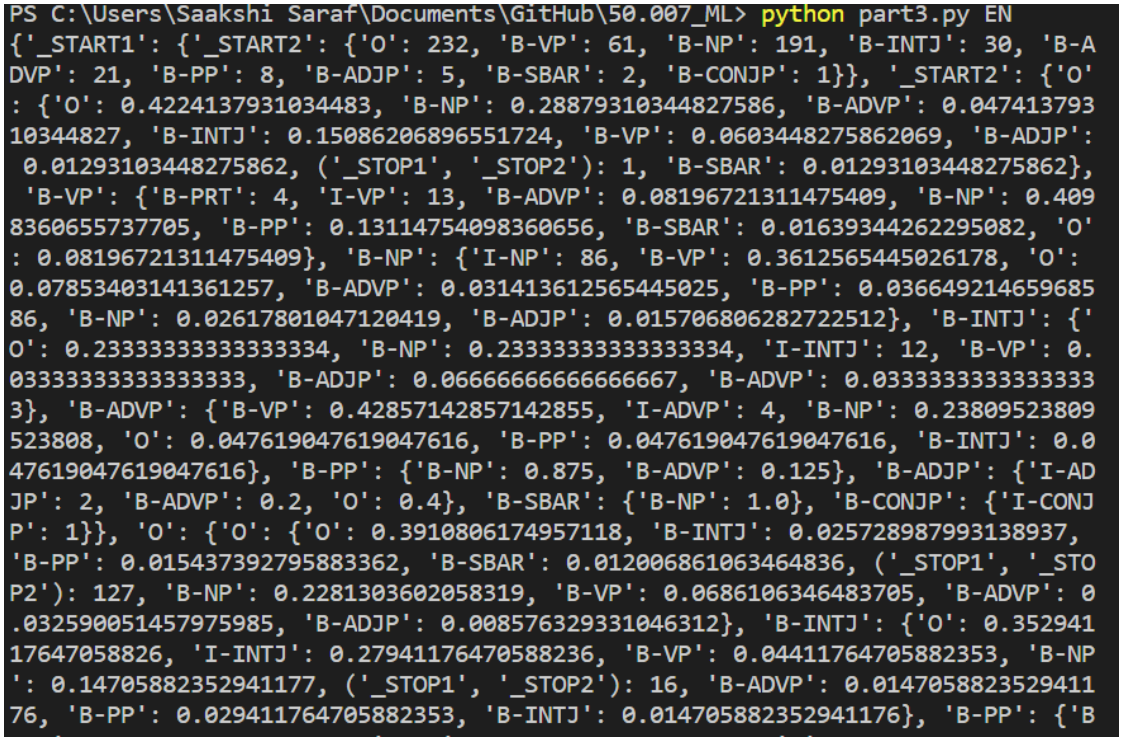
`Main():`
This function is responsible for inputting the necessary files and defining the output file which is to be written onto file dev.p3.out.

3.3 Algorithm and Model:

The model takes a training dataset containing sentences as an input and finds their corresponding tags. It then calculates the emission and transition probabilities for each word and tag pair, respectively. It also creates a set of unique words (using `UniqVocab():`) in the training dataset. The new Viterbi algorithm is then applied to predict the tag sequence for each sentence in the input dataset. Finally, the predicted tag sequence is outputted to a file. The difference this time being that it now takes second-order dependencies into account.

3.4 Results:

While we were able to successfully generate the transition probabilities for the second-order HMM model as shown below, we were unable to resolve a few errors in our Viterbi algorithm. Unfortunately, no effective results were produced as the errors in the Viterbi algorithm could not be solved when attempting to change the `Viterbi():` function which was necessary in this implementation based on the thought process as explained before. The transition matrix however is generated correctly and is as shown below:



Second-order transition probabilities stored in a dictionary

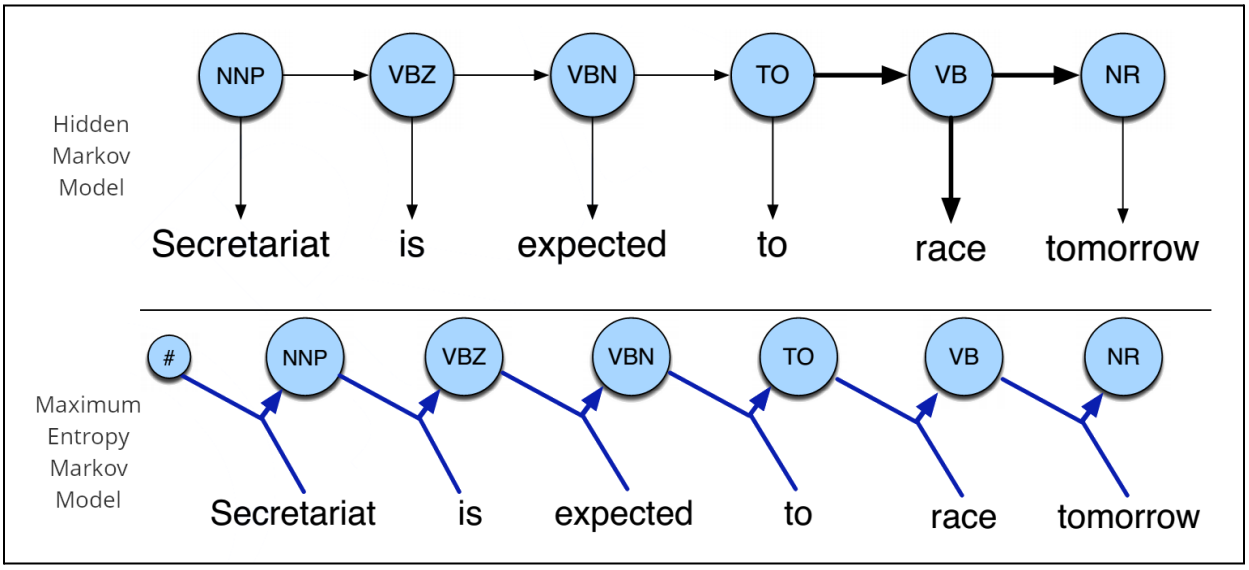
Part 4

4.1 Design Approach

The Hidden Markov Model in our previous implementations is a generative model. To improve upon the existing implementation, we explore discriminative models. One such model is the Maximum-entropy Markov model (MEMM), which combines features of Hidden Markov models (HMMs) and Maximum Entropy (MaxEnt) models.
After extensive literature review (refer to bibliography), we implemented the MEMM to improve the performance of the system in sentiment analysis of the given datasets.

4.2 Maximum-entropy Markov Model (MEMM)

In HMM, the current observation depends only on the current state and is assumed to be conditionally independent of all other observations given the current state. In contrast, MEMM allows the current observation to depend on the current state and also the previous observation:



HMM vs. MEMM for POS tagging

Another distinguishing factor which gives MEMMs an edge over HMMs is feature selection. In MEMMs, the emission probabilities are dependent not only on the current hidden state, but also on the input features for that time step. In our implementations, the input features we chose are : current word, the previous word and next word. Hence, apart from the emission and transition probabilities like in HMM, there are additional features of forward emission and backward emissions. We will explore how these additional probabilities work in the next section.
When calculating the probabilities in the modified Viterbi algorithm, we use a weighted combination of the emission, the forward and the backward probabilities.
To further improve the MEMM model, certain versions of the algorithm also take into consideration the second-order forward and backward probabilities. However, in interest of time, we have implemented only the immediate(first-order) forward and backward probabilities.

The MEMM model allows for more flexible modeling of the relationship between input and output, and can lead to improved performance as the input features are informative for predicting the output as we will see below.

4.3 Algorithm & Functions used

`discriminative_emissions()`

The emission probabilities in MEMM are defined as the probability of a token/word emitting a label/tag. As noted earlier, HMM is a generative model because words are modeled as observations generated from hidden states. Even the formulation of probabilities uses likelihood P(O|S), that is, emission probability of word sequence given tag sequence. On the other hand, MEMM is a discriminative model and directly uses posterior probability P(S|O); that is, emission probability of a tag sequence given a word sequence. Thus, it discriminates among the possible tag sequences and such probabilities are called “discriminative emission probabilities”. The emissions dictionary stores these probabilities in the format : The backward

emission c and forward emissions d contain dictionaries of the format:
{word:{tag: probability of word emitting tag}

This function also calculates forward and backward probabilities, in addition to the above mentioned emission probabilities. Forward emission probability is the probability of the next word at t+1 emitting the tag at t. Backward emission probability is the probability of the previous word at t-1 emitting the tag at t.
The dictionaries for storing these probabilities are of similar format to the emission probability
{word:{tag: probability of word emitting tag}

`Transition()`
The transition probabilities for the MEMM model are calculated in the same way as they were in the HMM implementation in part (ii). Hence we do not make any changes to this function

```
memmViterbi(emissionProbs, forwardProbs, backwardProbs, transitions,
weights, vocab, tags, sentence)
```
This is a modified form of the Viterbi algorithm. This function takes in the forward and backward emission probabilities in addition to the emission and transition probabilities. Weights is an array of values that are used as hyperparameters when calculating the temporary scores.

Initialization

Firstly, we initialize a dictionary called highscores, which keeps track of the highest scores and their corresponding parent tags for each position in the sentence. It initializes the score of the first position with "_START" tag to 0.0 and sets the parent tag to None.

Setting the scores:

The algorithm iterates through each word in the sentence and each tag in the tag set. For each tag, it calculates the score by considering the previous highest score, the emission probability for the current word and tag, the transition probability from the previous tag to the current tag, and the forward and backward emission probabilities. Each of these probabilities are multiplied with their corresponding weights (using the weights array). If the previous tag has no possible path to the current tag, or if the current word and tag are not present in the training set, it sets the score for the current tag to None and the parent tag to None. Otherwise, it sets the score and parent tag to the ones that resulted in the highest score.

Here too, to prevent underflow issues, we take the log of the probabilities instead of multiplying them together:

```
currentscore = previousscore[0] * weights[0] + log(a) * weights[1] +
log(b) * weights[2] + log(forward) * weights[3] + log(backward) *
weights[4]
```

The optimal values of weights are chosen through random trial and error, and by seeing which combination leads to higher accuracy. We define separate combinations of weights for the EN and FR datasets. Due to time constraints, we were not able to fine tune the parameters to find the optimal weights for best accuracy and hence resorted to random trial and error. This could be improved in the future by implementing grid search which allows for fine tuning of the parameters.

STOP case:

After iterating through the sentence, it reaches the end. For the STOP case, it considers the highest score for each previous tag and the transition probability from each previous tag to "_STOP". If there is no possible path to "_STOP", it sets the score for "_STOP" to None and the parent tag to None.

Backtracking:

Finally, it backtracks through the highscores dictionary to find the most likely sequence of tags for the sentence. It starts with the "_STOP" tag and follows the parent tag back through each position until it reaches the beginning of the sentence, adding each tag to the prediction list.

Overall, this algorithm aims to find the most likely sequence of tags for a given sentence by considering the probabilities of the emissions and transitions between each tag in the context of the entire sentence.

`ViterbiLoop()`

Similar to HMM, this function runs `discriminate_viterbi()` for each sentence in the dataset, and then finally merges the sequences of words and tags to write to the output file.

4.4 Results

For the given dev.in dataset, as expected, the MEMM model performs better than HMM, on both the datasets :

```
#Entity in gold data: 802
#Entity in prediction: 788

#Correct Entity : 562
Entity  precision: 0.7132
Entity  recall: 0.7007
Entity  F: 0.7069

#Correct Sentiment : 511
Sentiment  precision: 0.6485
Sentiment  recall: 0.6372
Sentiment  F: 0.6428
```

Prediction for EN

```
#Entity in gold data: 238
#Entity in prediction: 240

#Correct Entity : 158
Entity  precision: 0.6583
Entity  recall: 0.6639
Entity  F: 0.6611

#Correct Sentiment : 98
Sentiment  precision: 0.4083
Sentiment  recall: 0.4118
Sentiment  F: 0.4100
```

Prediction for FR

Bibliography

1. arvindpdmn, arpittrainer. (2022, February 15). *Maximum-entropy markov model*. Devopedia. Retrieved April 21, 2023, from https://devopedia.org/maximum-entropy-markov-model#McCallum-et-al.-2000
2. *Maximum entropy markov models for information extraction ... - MIT CSAIL*. (n.d.). Retrieved April 21, 2023, from http://www.ai.mit.edu/courses/6.891-nlp/READINGS/maxent.pdf
3. *Hidden markov and maximum entropy models T - Department of Computer Science*. (n.d.). Retrieved April 21, 2023, from https://www.cs.jhu.edu/~jason/papers/jurafsky+martin.bookdraft07.ch6.pdf
4. Cloud, A. (2018, May 18). *HMM, MEMM, and CRF: A comparative analysis of statistical modeling methods*. Medium. Retrieved April 21, 2023, from https://alibaba-cloud.medium.com/hmm-memm-and-crf-a-comparative-analysis-of-statistical-modeling-methods-49fc32a73586
5. *Maximum entropy markov models for Semantic Role Labelling - ACL Anthology*. (n.d.). Retrieved April 21, 2023, from https://aclanthology.org/U04-1015.pdf
6. E. Azeraf, E. Monfrini, E. Vignon, & W. Pieczynski. (2020, May 21). Hidden Markov Chains, Entropic ForwardBackward, and Part-Of-Speech Tagging . Retrieved April 21, 2023, from https://arxiv.org/ftp/arxiv/papers/2005/2005.08940.pdf
7. Named Entity Recognition: Concept, Tools and Tutorial. (2020, March 30). MonkeyLearn Blog. from https://monkeylearn.com/blog/named-entity-recognition/