50.007 Machine Learning - Design Project Report

Gan Jia Jie	1004113
Koh Jun Hao	1004295
Ng Yu Yan	1004334

General Approach For HMM, Part 1,2,3

We defined a HMM class to initialise the HMM emission and transition parameters to be used in the Viterbi and K-Best Viterbi algorithms which predict tags. The various parts will utilise these methods and we will reference them as we explain our approaches below.

Methods:

- **__init__(self,k):**
  Initializes structures to hold the possible tags, emission and transition parameters.
- **_count_all(self):**
  Iterates through all tokens and tags in the training dataset to populate the possible tags and emission and transition counts for the dataset.
- **_calculate_emission_MLE_UNK(self, x, y):**
  Calculates emission parameters based on training dataset, accounting for unknown words.
- **_get_argmax_y_part1(self, x):**
  Gets the tag that has the highest probability based on the given word, based on emission parameters alone.
- **_calculate_transition_MLE(self, prev_tag, tag):**
  Based on the given 2 tags/nodes, it accesses the dictionary transition_count to calculate the transition probability for the 2 given tags/nodes.
- **_viterbi(self, sentence):**
  Runs Viterbi based on transition and emission parameters, and returns the predicted sequence of tags.
- **_k_best_viterbi(self, sentence, k_num):**
  Runs K-Best Viterbi based on transition and emission parameters, keeping a list of k scores and paths, and returns predicted k^{th} best sequence of tags.
- **train(self, train_dataset):**
  Takes in the given training set and runs _count_all to obtain the transition and emission counts.
- **write_preds(self, filename):**
  Takes the predicted tags from Viterbi and writes an output file which has the tokens and its predicted tags, similar to the structure of the dev.out data sets.

Part 1

Approach

We first call **train(train_dataset)** to initialise the possible tags and emission parameters required. We calculated the emission parameters by keeping counts of all the times a tag appears (in self.state_count) and of all the times a tag emits a token (in self.emission_count).

We then called **predict_part1(test_dataset)**, which called **_get_argmax_y_part1(token)** for each token in a sentence, which called **_calculate_emission_MLE_UNK(x,y)** for each possible previous state/tag. **_calculate_emission_MLE_UNK(x,y)** calculates the emission probability for each tag and token/word by dividing the emission counts by state counts, as shown in the below equation.

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

_get_argmax_y_part1(token) then uses these emission parameters to get the most probable tag given the token.

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

We then write the predicted tags alongside each word to a file.

Results

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

Part 2

Approach

This part seeks to estimate and calculate the transition parameters for the HMM model. This is based off the following equation:

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

The various counts needed are obtained by using the **_count_all(self)** method. The code will iterate through the provided training set, counting the number of times the number of times each tag appears, appending to a dictionary called **state_count**. At the same time, it stores the previous tag, creating a transition pair and counts the number of times it appears, by appending into the dictionary **transition_count**. While iterating and counting,we used conditions to account for the different situations of the start and end of sentences.

To calculate the transition parameter/probability, the method **_calculate_transition_MLE(self, prev_tag, tag)**. The method takes in a given pair of tags, referencing **transition_count** to get the transition count and **state_count** to get the tag count, and dividing them to get the transition parameter/probability for this (referencing the formula above).

The Viterbi aims to predict the tags for a given set of tokens, by maximising the scores at each step as the program progresses in the forward direction down the sentence. The scores represent the joint probability of the input sequence and output sequence.

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

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

For our case, we decided to use the log of the scores:

$$\arg\max_{y_1 \ldots y_{n+1}} \log p(x_1 \ldots x_n, y_1 \ldots y_{n+1})$$

This is to prevent floating point errors as the scores/probabilities can get very small.
In our implementation of Viterbi, we predict tags sentence by sentence. We start by initialising a dictionary to store the scores calculated as the code iterates through the sentence, starting with the base case of:

$$\pi(0, v) = \begin{cases} 1 & \text{if } v = \texttt{START} \text{ (starting state, no observations)} \\ 0 & \text{otherwise} \end{cases}$$

We have a variable, **index**, to keep track of the position of the node we are analyzing in the sentence. We iterate through the tokens in the given sentence, to calculate and maximise the scores. Firstly, the token is used to calculate and retrieve the emission and transmission probabilities for every possible tag for the token. This is done through the

_calculate_emission_MLE_UNK and **_calculate_transmission_MLE**, which is then stored in a dictionary **state_scores**. We then proceed to calculate the log scores:

$$\pi(k,u,v) = \max_{w \in \mathcal{K}_{k-2}} (\pi(k-1,w,u) + \log q(v|w,u) + \log e(x_k|v))$$

$$bp(k,u,v) = \arg \max_{w \in \mathcal{K}_{k-2}} (\pi(k-1,w,u) + \log q(v|w,u) + \log e(x_k|v))$$

In the implementation of log in our scores, we had to write conditions to prevent log(0) errors.

After obtaining the difference scores, we take the best (maximum) of them, appending this best score and the tag that gives it to the score dictionary we initialised at the start. The code then proceeds forward with the next token with this best score until we terminate at 'STOP':

$$(y_{n-1}, y_n) = \arg\max_{u \in \mathcal{K}_{n-1}, v \in \mathcal{K}_n} (\pi(n,u,v) + \log q(\text{STOP}|u,v))$$

To obtain the sequence of tags, we make use of the best score from the last step previously calculated. The code will now iterate from the back of the sentence. Referencing the best scores calculated earlier and the transition probabilities/parameters, the code iterates through each possible state to find out which state gives the best score calculated.

$$y^*_{n-1} = arg\ max\ u\ \{\pi(n\ -\ 1,\ u)\ +\ log(a_{u,y^*_n})\}$$

The states are then appended and ordered in a list and the code will return this list.

Results

ES:

#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

RU:

#Entity in gold data: 461
#Entity in prediction: 533

#Correct Entity : 219
Entity precision: 0.4109
Entity recall: 0.4751
Entity F: 0.4406

#Correct Sentiment : 144
Sentiment precision: 0.2702
Sentiment recall: 0.3124
Sentiment F: 0.2897

Part 3

Approach

To find the 5th best output sequence, we modified Viterbi to store the best 5 scores and paths for each node (defined by position and tag) instead of the best score for each node. We generally used the same forward algorithm as Part 2 to calculate the scores of each possible path to the node, then put the 5 best scores and paths in the **state_score** entry corresponding to the node.

This gave us the 5 best paths when we reached STOP, and we are now able to return the 5th best output sequence.

Results

ES:
#Entity in gold data: 255
#Entity in prediction: 595

#Correct Entity : 112
Entity precision: 0.1882
Entity recall: 0.4392
Entity F: 0.2635

#Correct Sentiment : 75
Sentiment precision: 0.1261
Sentiment recall: 0.2941
Sentiment F: 0.1765

RU:
#Entity in gold data: 461
#Entity in prediction: 849

#Correct Entity : 211
Entity precision: 0.2485
Entity recall: 0.4577
Entity F: 0.3221

#Correct Sentiment : 115
Sentiment precision: 0.1355
Sentiment recall: 0.2495
Sentiment F: 0.1756

Part 4

Approach

For part 4, we were tasked to improve on the sentiment analysis system for tweets using any model we like.

As our current HMM model is a generative model, we wanted to try out an alternative model that uses a discriminative approach and compare their performance with each other. After reading up on multiple approaches such as the Maximum Entropy Markov Model, Conditional Random Fields, and Structured Perceptron. (Clouder, 2018) We have decided to use Structured Perceptron as our discriminative model of choice.

The Structured Perceptron Model is a discriminative model that is widely used in sequence tagging problems. This algorithm combines the Perceptron algorithm for learning linear classifiers with an inference algorithm such as Viterbi when used on sequence data. In a generative model, we usually calculate P(X|Y). In structured perceptron, feature representations are added to tokens, which are then used/referenced to provide a conditional probability P(Y|X) for decision Y given a history of X. (Here, Y refers to the tag and X refers to the token)

To run the Structured Perceptron algorithm (Collins, 2011), we can follow the approach below :

- Input: labeled examples, $\{(\underline{x}^i, \underline{s}^i)\}_{i=1}^n$.
- Initialization: $\underline{w} = \underline{0}$
- For $t = 1 \ldots T$, for $i = 1 \ldots n$:
  - Use the Viterbi algorithm to calculate

$$\underline{s}^* = \arg\max_{\underline{s} \in \mathcal{Y}} \quad \underline{w} \cdot \underline{\Phi}(\underline{x}^i, \underline{s}) = \arg\max_{\underline{s} \in \mathcal{Y}} \quad \sum_{j=1}^{m} \underline{w} \cdot \underline{\phi}(\underline{x}, j, s_{j-1}, s_j)$$

  - Updates:

$$\begin{aligned} \underline{w} &= \underline{w} + \underline{\Phi}(\underline{x}^i, \underline{s}^i) - \underline{\Phi}(\underline{x}^i, \underline{s}^*) \\ &= \underline{w} + \sum_{j=1}^{m} \underline{\phi}(\underline{x}, j, s_{j-1}^i, s_j^i) - \sum_{j=1}^{m} \underline{\phi}(\underline{x}, j, s_{j-1}^*, s_j^*) \end{aligned}$$

- Return $\underline{w}$

Fig 1: Structured Perceptron Algorithm Pseudocode

In the pseudocode shown above,

$$w := \ weight\ vector\ of\ features$$
$$\boldsymbol{\Phi}(x, s) := \ feature\ vector\ for\ token\ x\ and\ tag\ s$$

As the perceptron algorithm is a mistake driven one, the updating process will occur when the predicted tags do not match the actual tags. Hence, during the updating phase, the feature vectors for the actual tags will be promoted while the feature vectors of the incorrectly predicted tags will be demoted.

For feature extraction of the individual tokens, we made use of a wide range of features to create the feature vector of each word. These features are chosen based on the token characteristics such as its prefix and suffix, tags and previous tags, whether it is a capitalised word, etc. The image below shows us the full list of features that we defined for each token:

```
features = [
    f'PREFIX2_{prefix2}',
    f'PREFIX2+TAG_{prefix2}_{tag}',
    f'PREFIX2+2TAGS_{prefix2}_{prev_tag}_{tag}',
    f'PREFIX3_{prefix3}',
    f'PREFIX3+TAG_{prefix3}_{tag}',
    f'PREFIX3+2TAGS_{prefix3}_{prev_tag}_{tag}',
    f'SUFFIX2_{suffix2}',
    f'SUFFIX2+TAG_{suffix2}_{tag}',
    f'SUFFIX2+2TAGS_{suffix2}_{prev_tag}_{tag}',
    f'SUFFIX3_{suffix3}',
    f'SUFFIX3+TAG_{suffix3}_{tag}',
    f'SUFFIX3+2TAGS_{suffix3}_{prev_tag}_{tag}',
    f'WORD_LOWER+TAG_{word_lower}_{tag}',
    f'WORD_LOWER+TAG_BIGRAM_{word_lower}_{tag}_{prev_tag}',
    f'UPPER_{token[0].isupper()}_{tag}',
    f'TAG_{tag}',
    f'TAG_BIGRAM_{prev_tag}_{tag}',
    f'DASH_{"-" in token}_{tag}',
    f'WORDSHAPE_{self._shape(token)}_TAG_{tag}',
    f'ISPUNC_{token in string.punctuation}'
]
```

Fig 2: Feature List

For each epoch, the feature weights for the feature vectors will be referenced in the Viterbi algorithm to make new predictions. The model will learn from the incorrect predictions by updating its feature weights through the perceptron algorithm. The updated feature weights will then be used in the next iteration of Viterbi algorithm for prediction and this process will be repeated for the total number of epochs specified. Thereafter, the trained model will use the optimised feature weights to predict the output sequence of the test dataset.

Results

After training our Structured Perceptron model for **10 epochs** with a **learning rate of 0.2**, the training accuracy for the ES and RU train dataset are 0.946 and 0.957 respectively.

The trained model is then used to predict the output sequence of dev.in for both ES and RU languages. The output files are ES/dev.p4.out and RU/dev.p4.out respectively. Using the evalResult.py to compare the gold file and the prediction file yielded the following results:

ES:
#Entity in gold data: 255
#Entity in prediction: 154

#Correct Entity : 104
Entity precision: 0.6753
Entity recall: 0.4078
Entity F: 0.5086

#Correct Sentiment : 89
Sentiment precision: 0.5779
Sentiment recall: 0.3490
Sentiment F: 0.4352

RU:
#Entity in gold data: 461
#Entity in prediction: 506

#Correct Entity : 287
Entity precision: 0.5672
Entity recall: 0.6226
Entity F: 0.5936

#Correct Sentiment : 187
Sentiment precision: 0.3696
Sentiment recall: 0.4056
Sentiment F: 0.3868

For the prediction of the test datasets, we used the same trained Structured Perceptron model from above. It was trained for 10 epochs with a learning rate of 0.2 on the training dataset of the specified language. The trained model is then used to predict the output sequence of the test.in files. The predicted output files for ES and RU are named test.p4.out in their respective folders.

Conclusion

Based on the results obtained, we can see that a discriminative model such as Structured Perceptron performs much better than a generative one like HMM. The difference in results can be visualised in the tables shown below:

ES Dataset	HMM (Part 2)	Structured Perceptron
Entity in gold data	255	
Entity in prediction	551	154
Correct Entity	131	104
Entity Precision	0.2377	0.6753
Entity Recall	0.5137	0.4078
Entity F	0.3251	0.5086
Correct Sentiment	104	89
Sentiment Precision	0.1887	0.5779
Sentiment Recall	0.4078	0.3490
Sentiment F	0.2581	0.4352

Table 1: HMM vs Structured Perceptron (ES dev.in)

RU Dataset	HMM (Part 2)	Structured Perceptron
Entity in gold data	461	
Entity in prediction	533	506
Correct Entity	219	287
Entity Precision	0.4109	0.5672
Entity Recall	0.4751	0.6226
Entity F	0.4406	0.5936
Correct Sentiment	144	187
Sentiment Precision	0.2702	0.3696
Sentiment Recall	0.3124	0.4056
Sentiment F	0.2897	0.3868

Table 2: HMM vs Structured Perceptron (RU dev.in)

References

Clouder, A. (2018, May 11). HMM, MEMM, and CRF: A Comparative Analysis of Statistical Modeling Methods [web log]. Retrieved December 12, 2021, from https://www.alibabacloud.com/blog/hmm-memm-and-crf-a-comparative-analysis-of-statistical-modeling-methods_592049.

Collins, M. (2011, February). *The Structured Perceptron*. Retrieved December 11, 2021, from http://www.cs.columbia.edu/~mcollins/courses/6998-2012/lectures/lec5.1.pdf.