50.007 Machine Learning Design Project

Approaches and Results Report

Wang Zixuan (1004448)
Aditya Vishwanath (1004281)
Ramya Sriram (1004662)

In this design project, we would like to design our sequence labelling model for informal texts using the hidden Markov model (HMM) that we have learned in class. We hope that your sequence labelling system for informal texts can serve as the very first step towards building a more complex, intelligent sentiment analysis system for social media text.

Part 1

a) As instructed in the project description, we wrote a function, `estimate_e,` which uses Maximum Likelihood Estimator (MLE) to estimate the emission parameters from the training dataset.

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

`Estimate_e` takes the training data as its input and initializes the values `transCount` and `totalCount` to store the values of Count(y → x) and Count(y) respectively. It also initializes a dictionary e to store the two values as key-value pairs. The function then iterates over every sentence in the data, iterates over each word in each line, splits each line into x and y with `word[0]` and `word[1]` respectively since the data consists of the language word followed by the tag. For each time a given y_i transitions to a given x_i, the function updates `transCount[`y_i `,` x_i`]` as well as `totalCount[`y_i`]` by +1 each.

The function can then use these two values to calculate emission parameters as per the formula above.

b) Given the problem introduced in this part, which is that some words that appear in the test set do not appear in the training set, we need to make some edits to `emission_e` to counter it.
We first add the initialisation of the special wod token #UNK# as `unk_token = '#UNK#'` as well as `k=1` into the function. We then modify our calculation of emission parameters to:

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

The only changes we make to the current `emission_e` is that we alter our final formula to be the one above and after that, we iterate over all y's and execute the second formula for every y with the value `unk_token`.

c) We first format our dataset from files using `get_data` which takes filename as input to strip the data into sentences, then words, and then into token and label and append them into an array.

Following this, we defined a `predict_label` function which takes in filename, dictionary e as well as labels. This function then iterates over all y's, for known and unknown to compare and update the argmax, or `ystar` as we defined which is the value below, accordingly and output into the `dev.p1.out` file

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

We then implement a `eval_script` function which takes in prediction values and gold values as inputs and returns the comparison between the two using attributes of `evalResult.py`

We then evaluate our results with `evalResult.py`. Comparing our outputs with the gold-standard output in `dev.out`, we obtain the precision, recall and F scores as seen below:

ES	**Precision**	**Recall**	**F-score**
Entity	0.0588	0.3333	0.1000
Sentiment	0.0526	0.3190	0.0976

RU	**Precision**	**Recall**	**F-score**
Entity	0.0625	0.3333	0.1053
Sentiment	0.0601	0.3280	0.1011

Part 2

a) We used Maximum Likelihood Estimation (MLE) to estimate the transition parameters for the training set:

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

`Count_transition` counts the number of observations transition occurs from one state to another state. For each sentence with a token and tag, we stored the counts in a nested dictionary called `track_transition`, containing the state and number of observations.

Hence, we can now easily input the values to calculate the transition parameters, which is found with the function `transition_parameters`.

For $q(STOP|y_n)$,

$$q(STOP|y_n) = \frac{track\_transition[y_n]}{sum(track\_transition[y_n].values)}$$

For $q(y_1|START)$,

$$q(y_1|START) = \frac{track\_transition[y_1]}{sum(track\_transition[y_1].values)}$$

b) After estimating transition and emission parameters, we implemented the Viterbi Algorithm to compute the following:

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

To reduce the risk of potential numerical underflow, we calculated all values in natural logarithmic form (using addition instead of multiplication). This was beneficial as it allows us to take the first word in the sentence without taking into account the START position because $ln(1) = 0$.

When testing at every position, we check if the observation has appeared in the training set. If it is different, we assign a special token #UNK#. We also calculate the natural logarithmic values of the transition and emission probabilities. Using the Viterbi function we have implemented, we calculate the best path from START to STOP, retrieve the states and form the full tag sequence and write outputs in the file `dev.p2.out`.

We then evaluate our results with `evalResult.py`. Comparing our outputs with the gold-standard output in `dev.out`, we obtain the precision, recall and F scores as seen below:

ES	**Precision**	**Recall**	**F-score**
Entity	0.3835	0.4	0.3916
Sentiment	0.3158	0.3294	0.3225

RU	**Precision**	**Recall**	**F-score**
Entity	0.4029	0.3644	0.3827
Sentiment	0.2710	0.2451	0.2574

Part 3

For part 3, since we are to find the 5th best sequence instead of the best, we decided to modify the Viterbi formula so that we are only getting the 5th best. Recall for Viterbi algorithm, in order to find the most optimal sequence, we get the highest score from all of the previous nodes, added with the transition from that node to the current node. To think about the Viterbi algorithm, the best sequence is the one where the final state transitions to the stop state with the highest score of pi multiplied by the transition parameter between the 2 states. If we think of it like that, to find the 5th best sequence, we just have to find the 5th highest scoring between the maximum scoring sequence of the last node with the transition parameter between the last node and the stop state, in logarithmic form.

As such, we modify the last part of the sequence, and we find the number of transitions that can happen, and afterwards we create a shallow copy of the array of scores for the total score for the last node using logarithmic form by the transition parameter to stop state. We then sorted the array in ascending order and found the 5th best in the array as the one for this sequence. After this is done, we have to backtrack to find the sequence that we were looking for just like the Viterbi algorithm beforehand.

We could also use another method, whereby we store the 5 best sequences for each node. Instead of just storing the argmax of y, we could also store the parent as well as its order(which is which nth best sequence it belongs to), so that we know where it came from for easier backtracking, as well as the scores for each of the 5 best sequences for each node. So at each node, we have 2 arrays, array 1 containing a tuple with 2 elements, the parent node, and the nth sequence it came from in the parent node, and array2, the scores. The indexes are arranged such that the 1st entry of the 2 arrays are the best sequence to that node, and so on till the 5th entry in the array. At the last state before the stop state, we calculate the 5th best of all of the nodes combined, by getting the transition score and each of the 5th-best sequences for all the last states added together. We can then begin backtracking to find the parent node as well as the nth sequence it came from inside the parent node.

ES	**Precision**	**Recall**	**F-score**
Entity	0.1720	0.3608	0.2329
Sentiment	0.1364	0.2863	0.1848

RU	Precision	Recall	F-score
Entity	0.1684	0.3124	0.2188
Sentiment	0.1111	0.2061	0.1444

Part 4

Alternative models for sentiment analysis

Naive Bayes

Apart from using the Hidden Markov Model, sentiment analysis can also be done using the Naive Bayes classifier. Naive Bayes classifier is one of the fastest classification algorithms for data of huge sizes. As the name suggests, the Naive Bayes classifier used the Naive Bayes' theorem with the assumption of strong independence between features. This classifier is very efficient in performing Natural Language Processing or NLP tasks such as sentiment analysis through text.

Given 2 outcomes, C and D, the Bayes' theorem is as such:

$$P(C|D) = \frac{P(D|C)P(C)}{P(D)}$$

The theorem is used to analyse the membership probabilities for each outcome, like the probability that a given word belongs to either outcome, assuming C and D are "positive" and "negative" respectively. The most likely outcome, or class, is defined as the one with the highest probability. In our lessons, we learnt it as Maximum A Posterior, or MAP.

MAP is calculated by:

$$MAP = argmax\, P(C|D)$$

When we sub in the Bayes' theorem into this, we get:

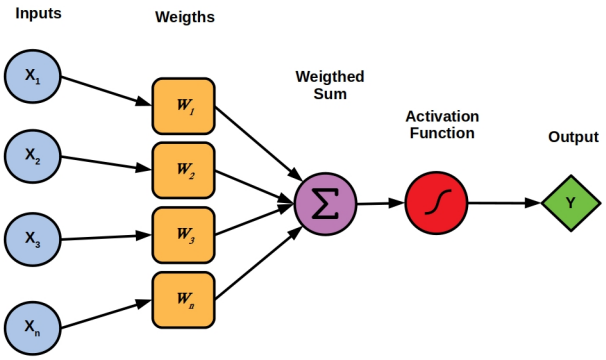
$$MAP = argmax(\frac{P(D|C)P(C)}{P(D)})$$

We could then adopt a Multinomial Naive Bayes classifier to perform sentiment analysis through text.

Perceptron

Another approach could be using a single perceptron to make a text sentiment classifier. The perceptron is a classifier we had learnt about early in this module and it represents the fundamental unit of a neural network.

Given a set of inputs x = $\{x_1, x_2, x_3, \ldots x_n\}$ and weights w = $\{w_1, w_2, w_3, \ldots w_n\}$, A single perceptron isas seen below where it takes in the inputs with their corresponding weights, w and b to produce and output Y ie [f(X) = w.X + b] that outputs [Y = f(X)]



A perceptron is trained by finding the optimal weights to predict Y. To do this, we would need to perform supervised learning using labelled data, which we are provided with, and also execute back-propagation to optimise the weights.

Back Propagation can be broken down into 6 steps:

1. Initialise w and b at zero or at random values
2. Make a forward calculation step to get sigmoid value (Y)
3. Calculate the cost function [(actual output - predicted output)]
4. Calculate dw and db [derivatives]
5. Calculate new w and b
6. Repeat step 1 o 5 until the cost function is closest to 0 or minimum

A function can be implemented which takes in:

1. Number of training units
2. Number of iterations
3. Learning Rate

4. Output Labels
5. Inputs

This function would then train the perc model with backpropagation and hence yield accurate results for predictions on which sentiment each word or text belongs to.

Reflections

Unfortunately, we were not able to implement part 4 of this project due to time constraints as well as our inexperience with the application of what we learnt in lessons into code. However, we were able to learn a lot about the various models that are used for sentiment analysis and their respective effectiveness.