50.007 Machine Learning
Group Project Report

Ang Yu Jie	1005270
Ho Wei Heng, Jaron	1005011
Nicholas Gandhi Peradidjaya	1005295

Part 1

Estimating Emission Parameters Naïvely

To find the emission parameters efficiently, we first create a dictionary of tags. For each tag, there is a corresponding nested dictionary that stores the total count of the tag and the count for each word associated with that tag. We will call this the emission table. We also construct a list of words that appeared in the training file. This is the `_construct_emission_table()` function (note the leading underscore).

Afterwards, we will be able to efficiently calculate the emission parameter by looking up the emission table for the relevant counts for a given tag and word. This is implemented in `emission()`.

Handling Unknown Words

To handle unknown words, when we encounter a new tag, we will initialise the nested dictionary with a count of 1 + *k* instead of just 1. We will also add a new entry into the nested dictionary for #UNK#, with a count of *k*. This is done using `construct_emission_table()` (with no leading underscore).

The emission table structure is still fundamentally the same, so we can reuse `emission()` to compute the emission parameters with the new table.

Simple Sentiment Analysis

This is done in the `sentiment_analysis()` function. We parse through the test file line by line to find the most probable tag for each word. Before computing the probability though, we need to replace words that are not found in our list of known words with the #UNK# token.

Then, we will construct a dictionary of tags with the respective emission probability. However, if the word is not found in the emission table, we will assign a probability of 0. Afterwards, we choose the tag with the maximum probability and construct a string with the word and the chosen tag, and add it to a list. Finally, we write this list of strings into the output file.

We caclulate the precision, recall and F scores by running the provided `evalResults.py` script. The results of our algorithm are as follows:

	EN	**FR**
#Entity in gold data	802	238
#Entity in prediction	1148	1114
#Correct Entity	614	186
Entity precision	0.5348	0.1670
Entity recall	0.7656	0.7815
Entity F	0.6297	0.2751
#Correct Sentiment	448	79
Sentiment precision	0.3902	0.0709
Sentiment recall	0.5586	0.3319
Sentiment F	0.4595	0.1169

Part 2

Calculating Transition Parameters

We compute the transition parameters in a similar way to the emission parameters. We use a nested dictionary to create a transition table, storing the counts for each state (tag) and the preceding state. We initialise the previous state as START, and parse the training file line by line until an empty line is reached. Since an empty line demarcates the end of a sentence, we will set the current tag as STOP, increment the counts in the transition table accordingly, then set the previous state as START and repeat the process with the next sentence. We also create a list of states found in the training file (thus omitting START and STOP). This is done in `construct_transition_table()`.

To get the transition parameter, we use `log_transition()` to look up the transition table and calculate the probability accordingly. To avoid numerical underflows, we return the logarithm of the probability instead of the probability itself. In the case where the previous state is STOP or the current state is START, we return negative infinity. Likewise, if the ordered pair of states does not exist in the transition table, we also return negative infinity.

Viterbi Algorithm

We follow the implementation of Viterbi algorithm in slides 22 and 23 of the lecture 15 slide deck. We begin by initialising a (n+2) x (m+1) matrix of 0s, where n is the number of words in sentence and m is the number of states. Using the diagram in the lecture slides as reference, we need n columns for each word of the sentence and 2 for START and STOP. We have m rows for the states in the training set, with an additional row dedicated for START and STOP states only. Finally, we set the (0, m) number in the matrix to be 1, as it is the START state.

We manually do the first and last iteration of the Viterbi algorithm since they involve the START and STOP states in the special row of the matrix. For the rest of the iterations, we loop through each state v in the current column j+1. Then, for each preceding state u in the j^{th} column, we sum up the transmission parameter, the emission parameter and the value stored in the (j, u) slot of the matrix. The largest sum (log-likelihood) is stored in the (j+1, v) slot of the matrix. For the last iteration, we omit the emission parameter as there are no words left.

Note that we use two helper functions here: `log_emission()` and `inf_sum()`. `log_emission()` works identically to `log_transmission()`, but is exclusively used for the emission parameter for clarity's sake. `inf_sum()` is used to avoid an issue where extremely large numbers are being used for calculation. It does so by returning negative infinity if any of its inputs is negative infinity, and the sum of the inputs otherwise.

With the probabilities computed for every word-tag combination, we begin the backtracking process to determine which tag is the most probable for each word. We calculate the log-likelihood in the same way as the forward process, but this time we take the argmax instead of the max of the log-likelihoods. Finally, we return the list of tags.

All of this is implemented in the `viterbi()` function. It is called by the `main()` function for each sentence in the test file, then uses the results of the `viterbi()` function to write to an output file. The results of the Viterbi algorithm is as follows:

	EN	**FR**
#Entity in gold data	802	238
#Entity in prediction	783	445
#Correct Entity	514	136
Entity precision	0.6564	0.3056
Entity recall	0.6409	0.5714
Entity F	0.6486	0.3982
#Correct Sentiment	449	76
Sentiment precision	0.5734	0.1708
Sentiment recall	0.5599	0.3193
Sentiment F	0.5666	0.2225

Part 3

2nd Order Hidden Markov Model

To produce a 2nd order hidden Markov model, we have to tweak three functions. The first one is `construct_transition_table_2()`, which now tracks two preceding states instead of one. The second is a helper function, `ttable_append_2()`, where instead of using the preceding state as the key for the transition table, it uses a tuple of the two preceding states as the key. This allows us to retain the structure of our transition table.

Finally, we also have to modify the Viterbi algorithm, which is now `viterbi_2()`. We now initialise the matrix with an additional column since we have an additional START state in the beginning.

In the iterations of the forward process, for the current state *v* in column *j*+2, we loop through each state *u* in column *j*+1 and calculate the "1st order probability" by summing the value in slot (*u*, *j*+1) of the matrix and the "1st order" transition probability from *u* to *v* (i.e. the same as in part 2, but without the emission parameter yet since it is unaffected by the number of preceding states to consider).

Then, we loop through each state *t* in column *j* to calculate the "2nd order probability" by summing the value in slot (*t*, *j*) and the "2nd order" transition probability from *t* to *v* via *u*. Then, we take the largest "2nd order probability" and add it to the "1st order probability" to get the total probability for state *u*.

After repeating this for each state *u*, we choose the maximum probability and add the emission parameter to it (unless it is the last iteration), then store it in slot (*v*, *j*+2) of the matrix. Note that since we are calculating both the "1st order" and "2nd order" transition probability, we have to construct two transition tables beforehand.

To backtrack, we follow the same process for calculating the probability, then take the argmax instead of the max.

The results of our 2nd order hidden Markov model are as follows:

	EN	**FR**
#Entity in gold data	802	238
#Entity in prediction	690	57
#Correct Entity	453	35
Entity precision	0.6565	0.6140
Entity recall	0.5648	0.1471
Entity F	0.6072	0.2373
#Correct Sentiment	408	19
Sentiment precision	0.5913	0.3333
Sentiment recall	0.5087	0.0798
Sentiment F	0.5469	0.1288

Part 4

3rd Order Hidden Markov Model

We considered several models like naïve Bayes and neural networks, but we were ultimately unable to implement them properly. As such, we decided to extend our hidden Markov model to the 3rd order. Even though the 2nd order model performed worse than the 1st order one, we believed that giving the model more context would enable it to predict the tag more correctly. Unfortunately, it ended up performing slightly worse for the EN test set and about the same for the FR test set.

The implementation is done similarly to how we extended the 1st order model to 2nd order in part 3. The results are as follows:

	EN	**FR**
#Entity in gold data	802	238
#Entity in prediction	642	56
#Correct Entity	419	35
Entity precision	0.6526	0.6250
Entity recall	0.5224	0.1471
Entity F	0.5803	0.2381
#Correct Sentiment	373	18
Sentiment precision	0.5810	0.3214
Sentiment recall	0.4651	0.0756
Sentiment F	0.5166	0.1224