**ML Project**

**Group members:**Ooi Kai Lue (1001779)  
Jesandry (1001642)

**Estimation of emission parameters**

Firstly, the program reads the training file line by line and the label is separated from the word. This is store in a dictionary with key (word, label) and its value is the number of times this combination appears in the file. After that, the program simply uses the formula:

to calculate the emission parameters.

Next, it stores a dictionary with key word and its value being the number of times the word appears in the file. If the word appears < k times (in this case 3), these words would be categorized under #UNK#. This is done by create new key-value pairs with key (‘#UNK#’, label) and its value the number of times this combination appears. Once again, the emission parameters are calculated with the above formula for the #UNK#. Any probabilities that are for the words that should be categorized under #UNK# are then removed.

**Simple Analysis System**

Using the formula:

we can create a simple system to predict the label a word should have. The program simply finds the values of the emission parameters for a word with any possible label and the maximum value and its corresponding label is chosen. If emission probability of a word for every label does not it exist, we use the probability of #UNK#. If at least one of the emission probabilities of a word exist, any word-label combination of that word that is not found would have probability 0.

**Estimation of transition parameters**

To calculate the transition parameters, we can use the formula:

The function has three states, namely ‘start’, ‘stop’ and ‘neither’. For each of the word in the file, we get the label and then check if it is already found in the transition parameters dictionary to be returned. If it is not, we will create a new entry or it in the dictionary. Else, the count will increase by 1. At the same time, we have a separate dictionary that keeps track of the total count of each label, which the count increase by one for the label found of each word. It begins with ‘start’ state, where the transition from “START” will be added to the transition parameters dictionary. From ‘start’ state, we go to ‘neither’ state which will repeat the process mentioned above unless the word is ‘’ (empty). This will bring us into the ‘stop’ state which will add the transition to “STOP” into the transition parameters dictionary. When we finished all the words, we take each possible transition state and divide by the total count of the origin state to get the respective probability. Finally, for each label which transition from “START” to it does not exists, we add it into the transition parameters dictionary with probability of zero.

**Viterbi Algorithm**

Our implementation of Viterbi Algorithm can be separated into several steps.

Firstly, we separated the emission for “#UNK#” from the emission parameters dictionary generated into another dictionary which will be used for word that are not found in the emission parameters dictionary. Then, we process the dev.in and make all the words into element for a list (sentence), and the list of words as element for an outer list (all\_sentences). Next, for each sentence in all\_sentences, and for each word in sentence, we first check if the word can be found in emission parameters dictionary. If the word is not found, we replace the emission parameters dictionary with the unknown emission parameters dictionary. With that, we begin the forward process using a for loop in the range of the length of the sentence, which is the total number of words, including punctuations contained in the sentence. The index of the for loop is used as the steps or the state or the level of the forward process. If the level is 0, the first step, for each possible emission of the word, we get the transition probability from “START” to the emission and multiply with the emission probability. We also add into the list (backtrace) of the label before the current emission. The index of backtrace corresponds to the steps as well. We also add the probability into current\_prob to keep track of the probability of each node. Next, if the level is not equal to zero, first we transfer the current\_prob into old\_prob and rest the current\_prob to empty. Again, for each of the possible emission of the word, we multiply the emission probability with the transition probability for each label from the old\_prob together with the probability up until that specific node. We save the label that has the highest probability and set it as the most probable label before the current one and store it into backtrace. We also store the probability into current\_prob. By the end of the for loop, which is the end of the sentence, we calculate all the probability of the transition from the last emission to “STOP” and find the maximum and store it in backtrace as the first node from “STOP”. During the process, there may be some transition that are impossible, we handled it by changing the label into “O” and then continue as per normal from there. Then, we flipped the backtrace over, starting from “STOP”, we go through each of the dictionary in backtrace and find the parent label of each label. We append each of them into the list to be returned which is prediction. Finally, we flipped prediction and each label which makes its index will correspond to each word in the sentence of the same index. Using the prediction and sentence, we write into the output file dev.p3.out.

**Max-marginal Decoding Algorithm**

The max -marginal decoding involves the implementation of the forward-backward algorithm.

**Forward probability:**

The forward probabilities are defined as:

Base case:

Recursive case:

The program first calculates the forward probability for the base case using the base case’s probability and stores it in a list (this list stores all the *alpha* of a word) and this list is then appended to another list (this list stores lists of *alphas* of a sentence). This process is repeated for the recursive case using the recursive case’s probability. The base case is simply taking all the probability of the transition probabilities that start from the ‘START’ state. For the recursive case, the program takes the required values as shown in the formula above. If the emission probability is not available, the program first checks if the word with any other labels exist in the emission probability dictionary. If it exists, the program simply moves on as this probability would be 0 as the value of would be 0. If it does not exist, it would use the relevant value from the #UNK# values. If the transition probability is not available, the transition probability equals to 0 and the program simply moves on as the value of would be 0.

**Backward probability:**

The backward probabilities are defined as:

Base case:

Recursive case:

The program first calculates the backward probability for the base case using the base case’s probability and stores it in a list (this list stores all the *beta* of a word) and this list is then appended to another list (this list stores lists of *betas* of a sentence). This process is repeated for the recursive case using the recursive case’s probability. The base case is simply taking all the probability of the transition probabilities that end with the ‘STOP’ state multiply with the emission probability of the current word with a certain label. For the recursive case, the program takes the required values as shown in the formula above. If the emission probability is not available, the program first checks if the word with any other labels exist in the emission probability dictionary. If it exists (this means the word appeared more than *k* times and is not part of the #UNK# pool), the program simply move on as this probability would be 0 as the value of would be 0. If it does not exist, it would use the relevant value from the #UNK# values. If the transition probability is not available, the transition probability equals to 0 and the program simply moves on as the value of would be 0. For the backward probability, the above idea is also used for the calculation of the base case.

After that we can use the formula:

to calculate the most likely label a word in that sentence would have. This is achieved through using a nested for loop. The program first loops through the list of *alphas* of a sentence. In each loop, the program loops through the list of *betas* of that word. If the label associated with the *alpha* and *beta* is the same, they would be multiplied and compared to a variable which stores the max value of ) of that word, replacing it if the new value is larger. While looping through the *betas*, the moment a label is found to be the same, it would break out of the *beta* loop as we know that for each word, there is only one possible *beta* value for each label. The max value, together with the label is then appended to a list which stores the max value and label of each word for a sentence. This list is then appended to another list which stores for every sentence of the file used. Finally, this list is return and the program uses this list to write to an output file.

To check if the forward-backward algorithm is working, we can use the formula:

Using this formula, if we can get the same value regardless of j (for each sentence), we can say the algorithm is working as intended.

**Max-marginal Decoding Algorithm**

**New emission calculation:**

For the calculation of the emission parameters, we first follow the same method as the one given. After the #UNK# component is done, any words that is placed under #UNK# is restored back but the #UNK# values are not changed. For example: ‘the’ appears less than k times. It is then placed back in the dictionary of emission parameters while leave the values of #UNK# as it is. During testing, any word that is never found during the training period uses the #UNK# values while those words that appear less than k times uses its own emission probability.

**Posterior-Viterbi Algorithm:**

Instead of the traditional Viterbi algorithm, we used the Posterior-Viterbi algorithm which makes use of posterior probability of in the Viterbi algorithm. The table below shows the change in the formula used.

|  |  |  |
| --- | --- | --- |
|  | Viterbi | Posterior-Viterbi |
| Initialization |  |  |
| Recursion |  |  |
| Termination |  |  |

where if the transition probability from *s* to *k* > 0 and if the transition probability from *s* to *k* = 0, ./

In simple terms, the transition probability used in Viterbi is replaced by {1 if probability > 0, 0 otherwise} and the emission probability is replaced by .

More information can be found here:

[https://arxiv.org/abs/q-bio/0501006](https://arxiv.org/abs/q-bio/0501006%20)

**Different sets of prediction for entities and sentiment:**

Firstly, we made use of the above techniques to produce a predicted output. Next, we separated the prediction for a words’ entity and sentiment. If the label for a word is ‘O’, both its entity and sentiment would be ‘O’. The prediction for both the entity and sentiment makes use of the same method above. After they are predicted, we compare if they are consistent (for example: entity prediction is ‘O’, we check if the sentiment is also ‘O’). If they are inconsistent, we use the prediction in the first predicted output, else we combine the entity and sentiment to produce a new output.

**Results**

**Part 2**

|  |  |  |  |
| --- | --- | --- | --- |
| EN | FR | CN | SG |
| #Entity in gold data: 226  #Entity in prediction: 1201  #Correct Entity : 165  Entity precision: 0.1374  Entity recall: 0.7301  Entity F: 0.2313  #Correct Sentiment : 71  Sentiment precision: 0.0591  Sentiment recall: 0.3142  Sentiment F: 0.0995 | #Entity in gold data: 223  #Entity in prediction: 1149  #Correct Entity : 182  Entity precision: 0.1584  Entity recall: 0.8161  Entity F: 0.2653  #Correct Sentiment : 68  Sentiment precision: 0.0592  Sentiment recall: 0.3049  Sentiment F: 0.0991 | #Entity in gold data: 362  #Entity in prediction: 3318  #Correct Entity : 183  Entity precision: 0.0552  Entity recall: 0.5055  Entity F: 0.0995  #Correct Sentiment : 57  Sentiment precision: 0.0172  Sentiment recall: 0.1575  Sentiment F: 0.0310 | #Entity in gold data: 1382  #Entity in prediction: 6599  #Correct Entity : 794  Entity precision: 0.1203  Entity recall: 0.5745  Entity F: 0.1990  #Correct Sentiment : 315  Sentiment precision: 0.0477  Sentiment recall: 0.2279  Sentiment F: 0.0789 |

**Part 3**

|  |  |  |  |
| --- | --- | --- | --- |
| EN | FR | CN | SG |
| #Entity in gold data: 226  #Entity in prediction: 164  #Correct Entity : 104  Entity precision: 0.6341  Entity recall: 0.4602  Entity F: 0.5333  #Correct Sentiment : 65  Sentiment precision: 0.3963  Sentiment recall: 0.2876  Sentiment F: 0.3333 | #Entity in gold data: 223  #Entity in prediction: 169  #Correct Entity : 114  Entity precision: 0.6746  Entity recall: 0.5112  Entity F: 0.5816  #Correct Sentiment : 73  Sentiment precision: 0.4320  Sentiment recall: 0.3274  Sentiment F: 0.3724 | #Entity in gold data: 362  #Entity in prediction: 212  #Correct Entity : 63  Entity precision: 0.2972  Entity recall: 0.1740  Entity F: 0.2195  #Correct Sentiment : 45  Sentiment precision: 0.2123  Sentiment recall: 0.1243  Sentiment F: 0.1568 | #Entity in gold data: 1382  #Entity in prediction: 911  #Correct Entity : 381  Entity precision: 0.4182  Entity recall: 0.2757  Entity F: 0.3323  #Correct Sentiment : 240  Sentiment precision: 0.2634  Sentiment recall: 0.1737  Sentiment F: 0.2093 |

**Part 4**

|  |  |
| --- | --- |
| EN | FR |
| #Entity in gold data: 226  #Entity in prediction: 175  #Correct Entity : 108  Entity precision: 0.6171  Entity recall: 0.4779  Entity F: 0.5387  #Correct Sentiment : 69  Sentiment precision: 0.3943  Sentiment recall: 0.3053  Sentiment F: 0.3441 | #Entity in gold data: 223  #Entity in prediction: 173  #Correct Entity : 113  Entity precision: 0.6532  Entity recall: 0.5067  Entity F: 0.5707  #Correct Sentiment : 73  Sentiment precision: 0.4220  Sentiment recall: 0.3274  Sentiment F: 0.3687 |

**Part 5**

|  |  |
| --- | --- |
| EN | FR |
| #Entity in gold data: 226  #Entity in prediction: 235  #Correct Entity : 126  Entity precision: 0.5362  Entity recall: 0.5575  Entity F: 0.5466  #Correct Sentiment : 83  Sentiment precision: 0.3532  Sentiment recall: 0.3673  Sentiment F: 0.3601 | #Entity in gold data: 223  #Entity in prediction: 227  #Correct Entity : 131  Entity precision: 0.5771  Entity recall: 0.5874  Entity F: 0.5822  #Correct Sentiment : 84  Sentiment precision: 0.3700  Sentiment recall: 0.3767  Sentiment F: 0.3733 |