Group 2 – Assignment 3 – CS4341

Mali Akmanalp – Peter Kalauskas – Thomas Liu  
 makmanalp peter.kalauskas kangchao

## Unigram/Majority Class

To start, we used unigrams to model the sequence. This was a simple matter of computing the most frequent symbol in the training set. Once this was done, the program would predict this symbol for everything in the testing set’s sequence.

## Bigram

The next step was to create to create a data structure for storing bigrams. We chose to use multiple hash-maps for this process. The first hash-map is the **prefix-map**; it stored **prefixes** (i.e. the first symbol of the bigram) as keys and **suffix-map** as the value. The suffix-map is hash-map whose keys are **suffixes** (i.e. the possible sequence of symbols following the prefix) and values are the number of occurrences of the concatenation of the prefix and the suffix in the training set. This entire data structure will be referred to as the **prefix-suffix-map**.

For bigrams, the prefix and suffix are both of length one. This made the implementation of the **creation-algorithm** (i.e. the algorithm used to create the data-structure) rather simple. The creation-algorithm simply iterated through the training set’s symbols, adding these symbols to the prefix-map when they did not already exist, and then peeked at the following symbol and added these symbols to the appropriate suffix-map or incremented the matching value in the suffix-map.

b

c

3

1

a

b

c

2

2

1

a

1

a

c

b

a

1

suffix-maps

prefix-map

Figure - the data structure after processing bigrams on the string abbabcabbac

Notice that in the example, the empty string is a key. This is because when the algorithm starts it has the empty string stored as the current string. This will result in the algorithm guessing ‘a’ for the first character in the test set.

## N-grams

Instead of hard coding a trigram model, we simply expanded our bigram model into an **n-gram** model. That is, our program would generate Markov chains of any length, *n*, from the training data. This was a simply a matter of modifying the system from the previous step to store prefix’s of length and suffix’s of length one. This model completely replaced the previous two models since it was able to do both unigrams and bigrams.

b

1

b

c

2

1

b

c

1

1

a

ca

ab

a

1

ba

b

1

bb

a

2

suffix-maps

prefix-map

Figure the data structure after processing trigrams on the string abbabcabbac

# Further Improvements

## Training During Testing

One easy solution to make the model more accurate is to add data to the prefix-map as answers to the test set are revealed. Since there is no uncertainty in whether or not a symbol is correct when it is revealed, we did not need to worry about smoothing our results, and this extra data could only improve our results.

## Back-off During Uncertainty (Challenge #2)

When running the n-gram algorithm, it is possible that the algorithm will encounter a prefix in the test-set that it never saw in the training set. In this case, the algorithm has no way to determine what the next possible letter could be, and it makes a random guess.

To fix this issue, we implemented a back-off system whereby if no prefix was found for the **current string**, the algorithm would look up a substring of the current string in the prefix table. This required all Markov chain lengths between 1 and n to be generated during the creation-algorithm.

This completely eliminated random guesses from the system, however, longer chain length were still weighted the most. This was problematic since different test-sets had the best accuracy using different n-gram sizes.

## Weighted Markov Chains During Training (Challenge #3)

To fix the issues leftover by the back-off algorithm we implemented a weighting system. This algorithm assigned each Markov chain a weight. When running on the test-set, the algorithm will check each possible current-string length, and find the Markov chain that has the highest certainty, where certainty equals . Basically, each weight determines how much a Markov chain should be trusted given its length. In general, short chains are susceptible to \_\_??????\_\_\_, and long chains are susceptible to over fitting the data. Nevertheless, each test-set requires different weight values to operate with the highest accuracy possible.

We used random walk to estimate the best weights during the creation-algorithm. First, half of the training set is processed and stored in the prefix-suffix-map. Next, weights are randomly assigned to each Markov chain length, and the accuracy is computed using the second half of the training set as a test set. After \_\_\_how many ??\_\_\_\_ random are tried, the creation-algorithm finishes by processing the remainder of the training set and storing this information in the suffix-prefix-map.

*One possible improvement that was never implemented would be to tweak these weights as the testing-set is revealed?????????????.*

# Accuracy

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm\Accuracy | **Trial 1** | **Trial 2** | **Trial 3** | **Trial 4** | **Trial 5** |
| **Unigram** |  |  |  |  |  |
| **Bigram** |  |  |  |  |  |
| **Simple Backoff** |  |  |  |  |  |
| **Weighted Markov** |  |  |  |  |  |

# How to Run