Simran Gidwani

Assignment 1

September 24, 2018

N-Gram Models

As I began thinking about how I would implement the unsmoothed and smoothed models for this project, I thought the best idea would be to write three separate classes. Each class would represent the n-gram size being either unigram, bigram, or trigram. As assumed, the unigram class was the easiest one to write. I began by reading in from the one of the training files and splitting each line into an array of characters. At first I thought using a linked list of nodes would be a good data structure by the fact that there is access to the previous node and the node that follows. However, due to the fact that the data would be in a key, value form where the key would be the pattern of characters and the value would be the number of times that pattern occurred in the test file, I decided that a hash map might be better. After reading all the words into an array of characters, I tested to see if the key already contained a value and if not, I put a 1 in place. If the key already did exist in the hash set, I just added one to the previous value for that key. I then wrote a method called calculate probability that took all the values in the hash map and divided them by the total number of characters in the test file and saved that as the probability.

After ensuring that the implementation of the unigram model was done correctly, I had to think about how I would adapt this code to read in two characters rather than one for the bigram model. I used the substring method in the string class to concatenate a word into a string of 2 characters. Then, same as the unigram model, I used a Hash map to store the keys and values although this time I used a hash map of char sequence instead of chars. After reading in the entire file and gathering how many times each sequence of two characters occurred in the file, I went on to the calculate probability method. At first I thought this would be fairly simple. However, instead of doing the probability in the form of C(A|B)/C(A), I did the C(AB)/C(Total # of sequences). After realizing this mistake, I went back and changed it. To do this, I created a Unigram Object in my Bigram class. This allowed me to access methods solely in the unigram class. I then wrote a get char method that accepted a character as a parameter. What this method does is, it takes in the character passed, searches the unigram model for that specific characters probability and returns it. That allowed me to get the probability of, for example, the sequence “ab” given that the first character was an “a”.

I knew that the implementation of the unsmoothed trigram model would be fairly similar to that of the bigram. As the previous two classes, I continued writing the same methods while adapting them to allow for char sequences rather than chars. I then began implementing the smoothing algorithms to my code. I began with the simplest one, the LaPlace model. I took my calculate probability method and adapted the formula to add one to the numerator and add the total number of char sequences to the denominator. I also made a variable named count that is passed in as a parameter as either a 0 or a 1. If it is a 0, it represents the normal unsmoothed trigram model. If count == 1, then this represents the LaPlace model. Essentially, it starts every value in the hash map at 1 instead of 0.

After getting the first smoothing method to work, I attempted the Linear Interpolation method. By this point in the program, I was able to get an idea of what methods I needed to call in order to get the information I needed. Since we already had to calculate the probabilities for the unigram and bigram models, all I had to do was multiple those probabilities by the lambda (1/3) and then add them together.

To incorporate the idea of reading in a test file, I made a user functionality that allows the user to enter in each of the training files one at a time. I then call the language models on that specific file and compare it to the data from the test file. In each of the unigram, bigram, and trigram classes, I wrote a section of code that reads in the input file and compares it to the data that is stored in the vocab of the test.txt file. If there is a character that is not the test file, it adds that character to the vocab hashset and add a 0 in the place of the value. I also added it to an int called unknown which at the end calculates the probability of unknown words in the test file. For all three models I also incorporated sequences that occur only one time in the unknown probability. I did this because, as learned in class, having a lexicon of unknown words is important. Since the test files could contain letters or two letters that are similar among languages, it is important to take into consideration how likely they occur.

By doing this, I was able to see that the probability of unknown words increased from unigram to trigram for each file as assumed. However, the most important factor to consider was which file among the three gave the smallest probability of unknown words. This is important to note because calculating this probability helped me determine what language the test file was written in. Although the English test file has a higher probability of bigram unknown words that the bigram in English, this might be because certain sequences of two are extremely likely in Spanish.

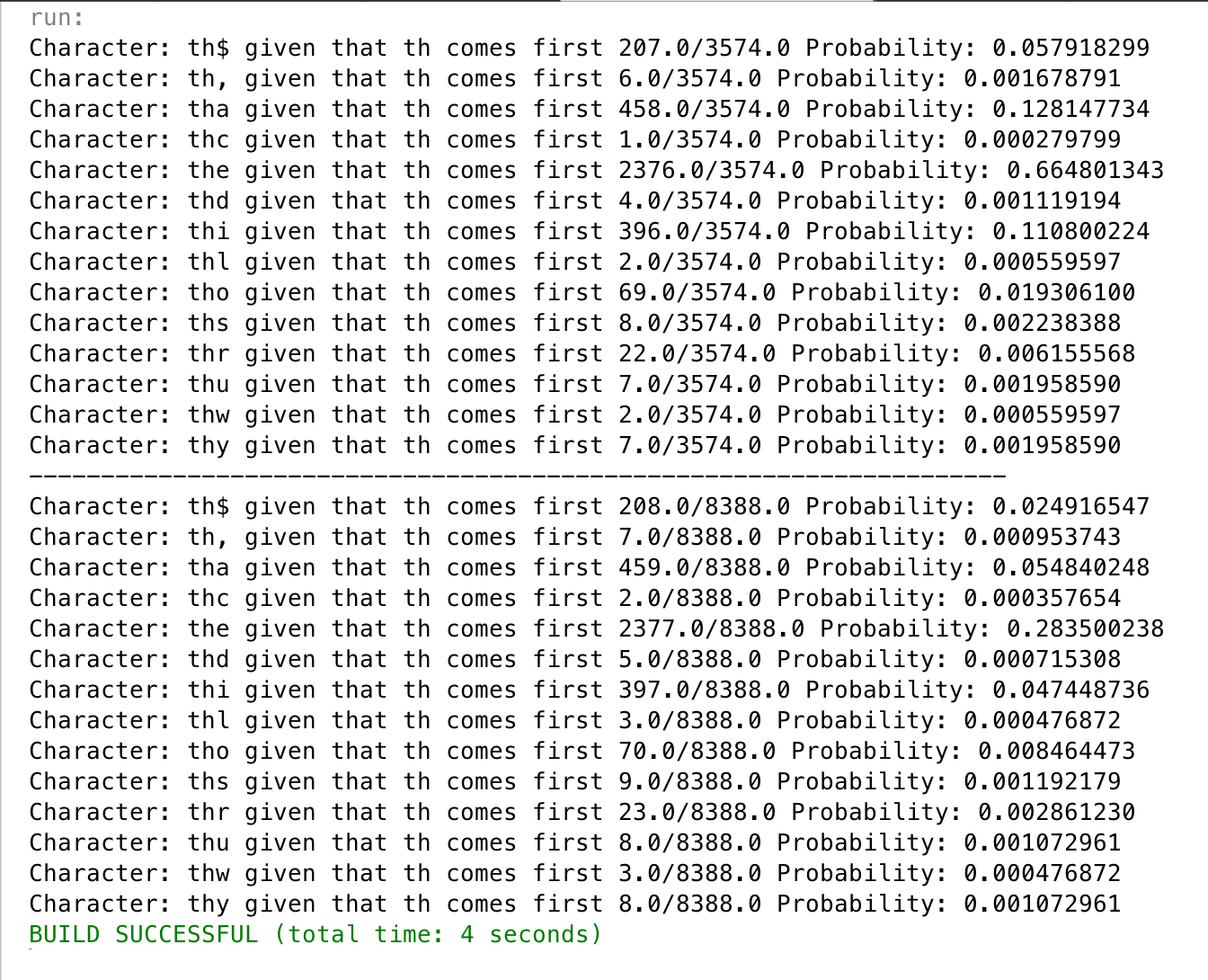
|  |  |  |  |
| --- | --- | --- | --- |
|  | Unknown Uni | Unknown Bi | Unknown Tri |
| Trainingde.txt | 8 | 504 | 3747 |
| Traininges.txt | 11 | 382 | 3380 |
| Trainingen.txt | 4 | 415 | 2337 |

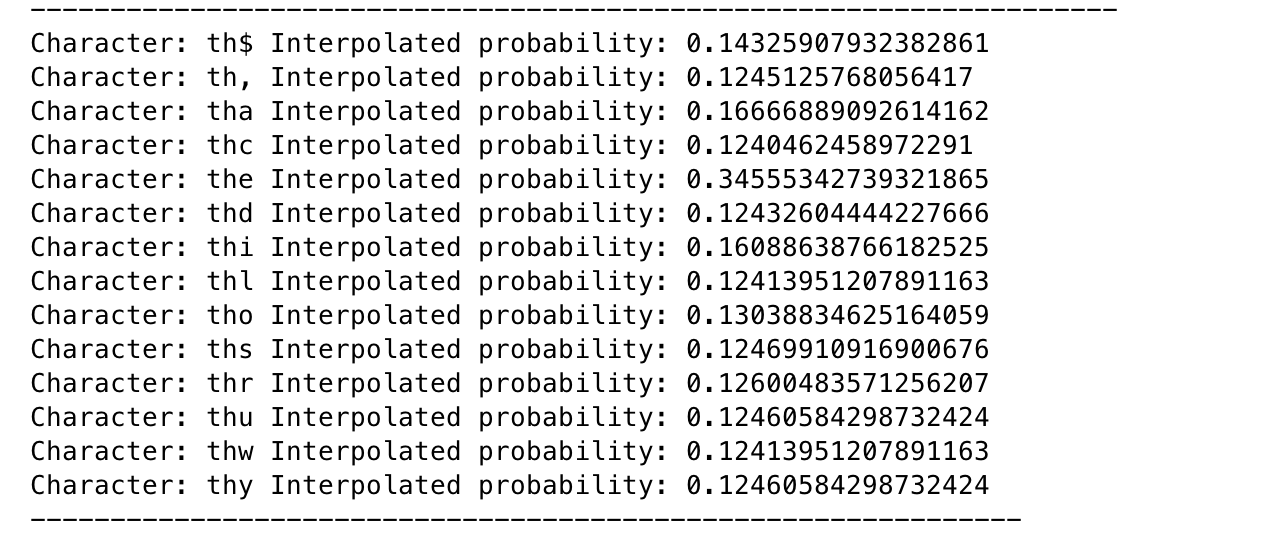
I then calculated the perplexity of the sentences. Since many of the probabilities were extremely small, it was important to calculate the log base 2 of the probability to get a higher value. This would make it easier to determine which language model and for what language, the perplexity was lowest. That was also another factor that helped determine what language the test file was written in. For the unigram, bigram, and trigram language models I was able to see that the training file in English has the lowest perplexities for each model. I was also able to look at the probabilities and see that they summed to one for the unsmoothed unigram model and the LaPlace smoothed for the trigram model.

Above dotted line represents unsmoothed trigram model

Below dotted line represents smoothed LaPlace trigram model

Third image represents the Linear Interpolation trigram model





Analyzing the perplexity scores for all the unsmoothed and smoothed language models tells you what language the test data is written in because as discussed in class, perplexity is also known as the weighted average branching factor. This represents the number of possible next words that can follow any word. By measuring perplexity you are able to see how often the same patterns of text occur in a test set compared to a training file. In this case, calculating the perplexity helped us determine that the language in the test file was most closely related to the language used in the training.en file which was English. Because perplexity is the inverse probability of the test set normalized by the number of words, minimizing perplexity equals maximizing the probability. Therefore, the lower the perplexity, the better the model.

Comparing the probabilities between the unsmoothed and smoothed scores of the trigram model help show that smoothed methods are generally more determinant when doing character based n-gram models. Since it was character based, the probability for actually predicting a language would be a lot lower that using words to predict the language. Although from my understanding of the smoothing models I believed that the Back-off and Linear Interpolation models would be fairly similar, it was interesting to see how they are represented in a different manner. As mentioned in class, the problem with the LaPlace smoothing method is that if the distribution size is either too big or too small, the probability will either increase by too much or decrease by too much when adding one.

Although the idea of the LaPlace smoothing method is to prevent assigning 0’s to probabilities of missing N-grams in the corpus, it does not truly reflect the real probability of each N-gram. While researching, I was able to see that this method is not commonly used. The reason it does not perform as well in some calculations is because much of the probability mass is moved to the unseen n-grams. Another method we were able to examine in this project was Linear Interpolation. This technique is considered one that focuses on the hierarchy of N-gram orders. It combines the probabilities calculated from all the N-gram orders. Using a Lambda value as 1/3 in this project, we equally weighted the probability of the trigram, bigram, and unigram for each sequence of characters. The last method we were asked to evaluate was the backoff method. The point of this model is to see whether the highest level N-gram is found and if not, then back off to the N-1 gram model and search there. Basically, if in the tri-gram model, the model does not return 0 as the probability of it’s occurrence, than you use the trigram model. However, if the model does return a 0, then it should back off and look for that same sequence in the N-1 gram model.

Just while going through and printing out all n-grams and their probabilities with the two letter history of “th” it was interesting to see that at a character level of just t, it would be extremely difficult to determine what part of the word the letter t was in. Just by comparing that to how often th was presented in the bigram model, it was easier to see where the characters “th” were represented in the word. Taking that to the next level in the trigram model, the number of possible representations of sequences with th as the first two letters decreased by nearly half. Comparing this data only makes it easier to understand why comparing larger sized n-gram models might make text more predictable or easier to identify. If I had been looking for a certain word in the test or training files, using a character based tri-gram model would be extremely inefficient due to the fact that many letters that appear next to each other in words are very common to begin with.