Michael Ing

ECE 2195 Hwk 1

Due 9/28/2018

Naïve Bayes Document Classifier

My Naïve Bayes classifier is composed of three .java files, entitled Words.java, Topic.java, and NB\_Runner.java. Words.java is the most basic of the files, defining a Word class that has a String for the word and an int to count the number of occurrences. This private Word class is contained within the Words class, which is simply an array of Words. The Word objects are stored in an ArrayList, which can be sorted via a method. The important methods in this class are contains(), which returns the logarithmic probability of finding a word in documents of this topic, and addWord(), which adds new words to the ArrayList or updates the corresponding Word’s count. For contains, if a word isn’t found then the method returns 1/(numWords+1) to avoid zero probabilities

The second file is Topic.java, which defines the Topic class. Each Topic object has a String name, an integer tracking the total number of documents it has, and a Words object that stores all Words that are found in documents of the topic. The main method in this file is addDoc(), which takes a file name and parses that file for words before adding them to the Words object for the topic. I chose to use several delimiters including a space, a comma, an exclamation point, forward and backward slashes, the at symbol, a period, an underscore, and a new line character to get distinct but common words, and not end up with a bunch of words strung together as with email addresses or contact info. I also removed all characters that weren’t a-z and put all characters into lower case, to get uniform data and get the most accurate representation of the documents. My final touch was adding a switch case statement that checked for common headers like path, date, lines, organization, etc., and either ignored those words or ignored the entire line following those words. This removed longer nonsensical words and lowered the word count in general to keep the process relatively fast.

The main file was NB\_Runner.java, which contained the main method and put everything together. This file defined constants such as the list of topics and the number of training and test documents to be used throughout the file. First the main method read in the training data, reading in a specified number of documents from each topic and adding the corresponding words to that Topics Words ArrayList. After reading in all of the words, each topic sorts the ArrayList in alphabetical order to make searching for words significantly faster. It then writes its list of words with frequencies to the data file for review, processed\_data.txt. After the training data has been properly parsed, the file then calls its testDoc() method, testing the remaining documents from each topic. In the testDoc method, an array of running products is kept to see which topic has the best fit for the current document. By parsing the document in a similar way to addDoc() from the Topic class, this file checks the corresponding probabilities of finding each word in each topic and updates the products array each time. It makes of the Naïve Bayes algorithm, multiplying by P(Y) (the probability of having a document in topic Y) and by P(Xi|Y) (the probability of seeing a specific word in a document of topic Y) for each word in the document. Since there are so many logarithmic probabilities being multiplied, products are bound to get extremely large. To account for this, if any product in the array would ever go past the maximum value, all products in the array for a specific document were reduced by a factor of 10^200. This tactic doesn’t affect the relative comparisons of the products but keeps them low enough to be comparable.

Once the products have been calculated, the file finds the minimum product and predicts the corresponding topic to be correct. It then checks this against the known topic, and writes its findings into processed\_data.txt, noting if it was correct or incorrect. At the end of the process, the file counts all correctly identified documents and outputs the % accuracy of classification. All data is stored in processed\_data.txt for debugging and further review.  
 At first accuracy was very low, sitting around 5%. I then changed the return value for the contains() method, noting that if a word wasn’t found it was currently returning 0. Rather than cause the whole product to go to 0 I changed this default return value to be a very small percentage, being 1/(total number of words + 1). That allowed the percentage to be small without zeroing the entire product. Using this tactic and a training/testing document size of 10 documents the accuracy jumped to 43%. I then added different delimiters other than a space, because I noticed that there were periods and @ symbols and other delimiting symbols that weren’t being properly utilized, which resulted in single instances of strange words like “aarondisgruntled”, and “aboutcounterselling”. I also added a switch case statement to ignore certain repeated lines for all documents, such as the number of lines or the date. I felt that these words added clutter and muddled the probabilities and could be removed to provide a clearer result. This caused the accuracy to jump to 51%.

It was around this time that I realized that my program had an extremely long runtime because I was searching unordered linked lists to find words. I realized that a sorted data structure would be much easier to search, so I changed my code to first use an array, and then an ArrayList of Word objects. By sorting these ArrayLists I was able to utilize the BinarySearch method, which drastically reduced the runtime and allowed me to increase my training/testing document size to 500 documents each. This sent the accuracy up to 81%. The final fix came from noticing that some words were still irregular as shown above, a combination of many common words. I discovered that this was being caused by new line characters with a lack of delimiters, so I added the new line character to the list of delimiters. This upped the accuracy to 83%, where it now remains.

After lots of deliberation and testing, I was able to improve the accuracy to 83% for my NB classifier. All relevant processed data is stored in processed\_data.txt and these three java files succinctly and effectively classify new documents in an efficient manner. I believe that with further testing and a bigger training data set, I could improve accuracy to make this system almost perfect.