Project 2: Naive Bayesian Spam Filter

Brandon Radosevich
New Mexico State University
EE 565: Machine Learning

September 15, 2016

Introduction To Bayesian Spam Filtering:

With the introduction of the second project, we were tasked with implementing a Spam Filter using Bayes' Theorem. In order to perform this task, one must fundamentally understand Bayes' Theorem for the probability of an event occurring based on related conditions.

Bayes' Theorem:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

In this project, this theorem is the basis for predicting whether an email should be tagged as Spam or Ham. This theorem states that the probability of A occurring given B is equivalent to the probability of B occuring given A multiplied by the probability of A divided by the probability of B. Now, there are four parts to this theorem, and each must be discussed in some detail. These parts are: **Posterior:** $= P(A \mid B)$, **Likelihood** $= P(B \mid A)$, **Prior:** $= P(A)$ and **Normalization:** $= P(B)$. For this project, the most crucial portion of this equation is the *Likelihood*.

Design Approach:

In the *CompanionFiles2*, there were over 70,000 emails for training. I began my initial design by spending a decent amount of time looking at the contents of both the Spam and Ham emails, and discovering the similarities and differences between both types of emails. From here, I began writing the the initial parsing code.

Initial Design Approach

Parsing: I began a somewhat blind attempt at parsing the emails by splitting each word in the email into a giant array. I did not implement any form of Regular Expression Handling, but rather I just loaded each string into a dictionary.From here, I would call my *countOccurrences* function which would count all occurrences of each word in a given email. I would then repeat this process N times, where N is a subset of the training file. Then, this would dump each word into a dictionary, where the word is the key, and

a tuple of (ham, #,spam, #) was the value. However, this proved to be very slow and time consuming.

Testing: As mentioned above, training using this parsing technique took 3 days to train on the entire set of training emails. Without implementing any Regular Expression Handling or limiting the length of dictionary keys, the Spam Filter was only 58% accurate.

Design Revisited:

Optimization: After being quite unsatisfied with the performance of my code, I decided to spend a few days optimizing the overall training process. I began by creating two separate dictionaries to hold the *SPAM* [word] = count and the *HAM* [word] = count. I decided to use three different techniques for saving the training files. When I combine the two dictionaries, I implement two versions of basic *Set Theory*.

Union:

$$\text{Union of } H\&K = (H_k) \cup (S_k)$$

Difference:

$$(H_{\text{kdiff}}) = (H_k) - (S_k)$$

$$(S_{\text{kdiff}}) = (S_k) - (H_k)$$

$$\text{H = Ham Dictionary , S = Spam Dictionary , and k = keys}$$

I began by taking all keys that occurred in both a the *HAM* and *SPAM* dictionaries and combing the values in a tuple. Then I took the difference of each dictionary and added that to the master dictionary, this was **Technique 1**. **Technique 2** utilizes the union of the two data sets. The thought here was only shared words between spam and ham email will provide the most information about whether the email is *SPAM* or *HAM*. Finally, the third technique I used was only taking N number of words from both dictionaries and simply write those words to the master dictionary. After using these three techniques, I brought my run time down two under two minutes.

Code Implementation:

I believe that I learned the most about *Python* and interacting with a large dataset from writing the Email Parser. As mentioned in the *Design Approach* section, my Email Parser initially took three days to train. After much Googling and recalling my Data Structures and Algorithms class, I was able to train on the 70,000 emails in under 5 minutes

emailParser.py

Data Structure After finding the built-in *dictionary* in Python to be rather slow, I stared researching better data structures for counting word frequencies in multiple text files. I came across a StackOverflow which mentioned the use of *Collections* for storing key,value pairs.[1] The feature I used the most in my code was the *Counter()*.

Algorithms In my Algorithms class, we learned that when you have a large data set it is best to implement a **Divide & Conquer** strategy for quick processing. [2] I implemented this algorithm by dividing the data set into Ham Training and Spam Training. Instead of having a master dictionary that has to count both Spam and Ham frequencies, I utilized two dictionaries to separate the count frequencies. Then after the training is completed, the two dictionaries are merged into a master dictionary, which is in turn written to a text file. In Addition to implementing a version of Divide and Conquer, I also implemented *Multi-Threading*. In order to reduce training time, I run both the Ham Training and Spam Training simultaneously.

classify.py The evaluation stage of the program is conducted in this portion of the code. Here, the user provides the recently created training file, as well as the evaluation file in order to predict whether each email is *HAM* or *SPAM*.

[1] **Python Collections:** `https://docs.python.org/2/library/collections.html`

[2] **Divide And Conquer:** `https://www.tutorialspoint.com/data_structures_algorithms/divide_and_conquer.html`

Simulations:

During the simulation stage of this assignment, I utilized three different dictionaries for training. Every time I trained on the data set, I saved three different versions of the training file. As mentioned in the *Design Revisited* section of this paper, I had three different techniques that I used for my system training. I found with *Technique 1* as well as *Technique 2* that my results were in the mid 90's for accuracy. Furthermore, I found that only taking N number of the most common words from the two dictionaries provided the least accuracy, leveling out around 70%. In addition, I found that tweaking either prior by more than $\pm$ 5% started giving bias.

System Training:

During the tweaking stage of this project, I began looking into different possibilities for increasing the accuracy. Once I reached 90 % accuracy, I really became interested in closing the margin between my code and the record. I began researching standard practices for Bayesian Spam Filters, and how they parse emails for key words. One night, I had a wild hair to Google python email parser. This is where I started to improve my results. This class, parses the email into a huge array, getting rid of non-ascii characters, and most importantly, those pesky viruses. After implementing the Python Email Parser library, I began playing with the word length of the dictionary as well as the the word parser. The last thing I implemented was only allowing words in the English alphabet into the master dictionary. Changing these three things drastically raised my accuracy from 90% to 97.6% in the test cases. My end parameters were as follows:

$$1 - 10 \text{ Word Length}$$

$$[a - zA - Z] \text{ Acceptable Characters for Dictionary}$$

Test Performance:

During the test performance, I used *CompanionFiles2/test1.idx* and *CompanionFiles2/test2.idx* for evaluating the overall efficiency of my program.

Test1:

Overall Accuracy: 0.973%
Confusion Matrix:

	Predicted	
Actual	Ham	Spam
Ham	1943(0.98%)	39 (0.02%)
Spam	81 (0.03%)	2547 (0.97%)

Average Word Match In Each Email: $\mu = 139.51$ words

Test2:

Overall Accuracy: 0.974%
Confusion Matrix:

	Predicted	
Actual	Ham	Spam
Ham	1888 (0.98%)	39 (0.02%)
Spam	79 (0.03%)	2604 (0.97%)

Average Word Match In Each Email: $\mu = 133.12$ words

Conclusions:

In Conclusion, this assignment was a great exercise in not only Naive Bayesian Probability, but learning how interact with large data sets in a programming environment. If time permitted I had a few more techniques I would have been interested in trying, however I am happy the results of my code. Please visit my *Github* account located at the bottom of this email for the full code. [3]

[3] **GitHub:** `https://github.com/bradosev03/NaiveBayesianSpamFilter`