Adaption of Statistical Email Filtering Techniques

David Kohlbrenner

IT.com

Thomas Jefferson High School for Science and Technology

January 25, 2007

Abstract

With the rise of the levels of spam, new techniques were applied for sorting email. Many of these were based off Bayesian statistical analysis. These techniques have been refined for the sorting of a very small fraction of pertinent documents from a much larger corpus of unwanted documents. A very similar situation exists in some expanding situations. A these corpora will contain tens of millions of documents not related to the current situation, and a few thousand at best that are pertinent. The adaptation of the spam filtering techniques to these situations is a relatively simple one.

Keywords: email filters, Bayesian

1 Introduction

There is a current rising need for finding documents important to certain situations in a large, and unsorted corpus of electronic documents. The current system is to have a small number of people read every single document and decide if it is pertinent. These cases often involve tens, if not hundreds, of millions of emails alone, and it is almost impossible to accurately review them all by human. There are current tools designed to help manage and sort these massive corpora, but they are all based on naive keyword searching. This costs both time and money, and often does not produce the needed result.

The goal of this project is to adapt the techniques devised for sorting spam from real email to other contexts. Spam is a very specific use of supervised learning as a sorting system, and it can be just as valuable in other contexts. In this paper I will explain the algorithms that have been created or adapted for use in email filtering, and the abilities of each. Also, I will walk through the parts of the filter that was built, what they are built with, and how they interact.

Over the past few years the level of spam present in email has risen dramatically. To combat this, statistical (guided learning) filters have been developed, and refined for sorting email. These filters train on email that has been sorted by a human (thus, guided learning) to learn what characteristics make a spam email 'spammy.' As the user classifies more and more email the filter becomes more and more accurate. A correctly designed and trained filter can exhibit accuracy of 99.9% in real world application. Many large email providers use

such filters today, but the exact implementations are usually secret. These filters have been specialized for one area of classification, but hold promise that their structures will work in other applications.

2 Background

Most email these days goes through a filter. The filter may be very simple, and look for a few keywords as defined by the programmer. Or they may be very complex, and over time the filter will learn the patterns and language choices spam has to make to get across a message. For many years, the prevailing method was to assign values for words, and then generate a score for an email. The problems with this are many-fold; first, a person has to classify words, and may miss key words or formatting clues. Second, as spam is blocked, spammers develop new ways around it, and a never-ending arms race between the people assigning word values and the spammers develops. Thirdly, the last few percentage points of accuracy are basically unachievable with hand made classification systems.[1] The exploration of statistical filtering for spam began in 1998 with Microsoft,[4] but it was not expanded upon until 2002. Paul Graham wrote an article in 2002 entitled "A Plan for Spam" in which he outlined his struggle with spam, and the abilities of statistical Bayesian filtering. Since then, development of statistical filters has moved forward, and is now an industry standard; reaching accuracy rates of 99.9% [3] Unfortunately, a plateau occurs at that mark, and the fourth nine of accuracy is very difficult to get. A number of strategies have been created for this; from Markovian classification, to intelligent feature set reduction.[2]

A Bayesian filter is designed to do two things; train on categorized email, and categorize new email. The first thing done with a filter is training. Initial data, categorized as spam and non-spam (ham) must be fed into the filter so it has some starting point.

The first thing a filter has with an email is tokenization, or feature set creation. A simple filter uses whitespace or punctuation as delimiters, and everything else is a token. One of the first advances was to make sure HTML and header files were being included in the tokenization step. These aspects can turn out to be just as valuable as any specific English word.[1] A good way to improve accuracy of a filter is to allow it to use tokens representing things not easily understood by a human. Chained words (phrases), identifying patterns in those phrases, and Sparse Binary Polynomial Hashing are some of the methods that have been used.[2] Chained words are just that, words put together as a phrase and considered one token. A good example of this would be the phrase "free day." Suppose you tokenized this with simple whitespace delimiters; you would get the tokens "free" and "day." Probably "free" would show up many spam emails, and would therefore be considered spammy. "Day" seems relatively neutral, so the over all outcome of the phrase would be spammy. With chaining, "free" and "day" stay the same, but the token "free day" is introduced. If you often discuss when you will be available with friends, the phrase might have a very high 'ham', or legitimate, value associated with it.[2] Sparse Binary Polynomial Hashing (SBPH) is a conceptual extension of chaining. With SBPH, you generate phrases which contain a list of ordered tokens in them. The phrases are treated much the same way tokens usually are, and a probability of the phrase being spammy is derived. Where SBPH is different is in

analysis of a new email. The phrase is compared to all possible same length phrases in the new email. Depending on the number of matches of positions of words, the phrase is given a larger weight in determining spammyness. These techniques, along with a few others have come to be known as "concept identification."[2]

The next step is to train a filter. This consists of creating a database of the tokens created during the tokenization phase and then storing their occurrences in user categorized email. Once that is complete, the filter must generate a probability that any given word is spammy. The most basic P algorithm is Paul Graham's; which is $P = \frac{(AS)/(TS)}{((AS)/(TS))+((AI)/(TI))}$ where AS and AI are the total appearances in spam and innocent email, and TS and TI are the total number of spam and innocent emails in the corpus. An improvement to this by Gary Robinson included accounting for the amount of information that has been seen. Robinson's is $F = \frac{SX+N(P)}{S+N}$ where S is a tuning variable, N is the total appearances of the token, P is Graham's value for the token, and X is the hapax value.[2] This brings us to hapaxes, or single-corpus tokens. These are tokens that appear in either spam or innocent email exclusively. Because you have no precedent for a hapaxial token, you must assign an arbitrary value to them. According to most papers[1, 2, 3] the best value is a 40% chance of being spam. This is because if you have not seen a token, it is probably not spam- the massive amount of spam means that most tokens in it have been seen.[2]

There are three methodologies as to what you train after the initial training set is complete. The first is train all user flagged messages, or TEFT. The second is train on error, or TOE. TOE means that if the system prediction does not match the user classification the message

is trained, but only in those circumstances. The last is train until no errors, or TUNE. TUNE is the same as TOE, except the message is trained and then analyzed until the prediction matches the user classification. The advantage of TUNE is that new spam techniques are learned almost instantly.[1, 2]

Once a message has been tokenized, and a trained filter is in place, it can analyze a message. The filter uses one of a many algorithms for statistical combination. The original is Bayesian Combination, which is $\frac{AB}{AB+(1-A)(1-B)}$. Where A and B are the two probabilities to be combined. This method requires that the developer put a pre-set limit on the number of examined tokens. Robinson wanted to allow for more than the two extremes Bayes allowed, and created Robinson's Geometric Mean Test, which outputs three values, not just one. P the level of spamminess $= 1-((1-P_1)(1-P_2)...(1-P_N))^{(\frac{1}{N})}$, Q the level of nonspammyness $= 1-((P_1)(P_2)...(P_N))^{(\frac{1}{N})}$ and S the combined indicator $= \frac{1+\frac{P-Q}{P+Q}}{2}$. The final, and most advanced algorithm, is Fisher-Robinson's Chi^2. This was an advance released by Robinson based on the work of Sir Ronald Fisher. It is very similar to Robinson's earlier algorithm, but uses an inverse Chi^2 function to create the P and S values. Once an aggregated value is calculated from one of these, it is compared to a threshold value, and then a system classification is set. [1, 2]

3 System

The final product is a modular filter designed for working in a non-dynamic corpus with dynamic classifications. It consists of Java packages, designed to work together. Because

this system was started with only a few weeks of available development time, tokenization, and the feature extraction techniques that go with it, were not developed for this project beyond the conceptual stage. This work will be done separately, and will not impact the inner workings of the filter, just the accuracy of its output.

3.1 Initial Research

To create the filter, I read the book "Ending Spam" by Jonathan Zdziarski, and developed a set of features that would be useful in our context. The system was then drawn out on a white board and the configuration of the parts was decided upon. During coding, minor structural changes were made. The entire project was written in the Eclipse Java 1.5 environment with SQLite 3 as a database system. For keeping all code up to date among different users, we used a local Tortoise SVN server. The project went through the standard waterfall life cycle, due to time restraints it was not iterative. As each of the packages was developed, they were tested using a set of simple tests that were created as needed. Systems testing was more comprehensive, but could not begin to occur until all parts were operational.

3.2 The Database Package

The first package to be developed was the database package, as all the other packages would need to refer to it to accomplish anything. The original model for the database package called for 4 components; a token class, a message class, an interface for a token database, and an interface for a message database.

3.2.1 The Token

Consists of an ID number (a Long), a count of its appearances in relevant emails, a count of its appearances in irrelevant emails, and the current 'P' value, or probability that it is irrelevant. I decided to keep all actual tokens in the pre-processing tokenization phase, as opposed to continuing to use them within the databases. The reason for this was two-fold: first, this ensured that all feature extraction took place outside of the program, and second, some of the tokens could be very complex and take up large amounts of space.

3.2.2 The Message

Consists of a message ID number (a Long), the current user classification, the current system classification, and an unordered list of the ID numbers of tokens contained in the message.

3.2.3 The Message Database

There are two versions of this, one for interfacing with a SQLite database, and one for storing all data in memory. In the original design, all database work would be done with persistent SQLite databases. An open source JDBC driver for SQLite was used for interfacing with the actual databases. The SQL version has a single table with message ID as primary key, and the other containing current classifications. There is also a second table that contains message IDs mapped to all the tokens they contain. The memory version was developed during testing, when it became apparent that the SQL accessing would take longer than was practical for testing. The memory system is a hash map with the key as the message ID and the value as the associated message object.

3.2.4 The Token Database

Again, there are two versions of this, one for interfacing with a SQLite database, and one for storing all data in memory. The SQL version contains a table with token ID as primary key, and the other aspects of a token as data. The memory version uses a hash map with token ID as the key, and the associated token object as the value.

3.3 The Training Package

The second package is the training package. Again, the goal was modularity for the sake of later modification. At the time of writing the package consisted of two P value algorithms, a trainer, and a training data system.

3.3.1 P Value Algorithms

These were discussed above in background. Both Graham's and Robinson's methods are included here, and are designed so they are identical to the other parts of the system. Each takes a token, and calculates a probability that the token is irrelevant. That value is returned to the analyzer for assignment to the token.

3.3.2 The Trainer

The trainer itself takes all of the user classified emails, and trains the databases on that data. As discussed above in training methods, there is room for different modules for different training methods. At the time of writing, only a simple train all messages once system was complete, but a TUNE or TOE system was planned. Once it has updated the database

with all the instances of tokens, it calls a P value algorithm to determine the P values for each token. Future versions might then run a preliminary analysis, and train messages again based on the results.

3.3.3 The Training Data Interface

This was created partway through development as a mediator between the trainer and the databases. During training, billions of edits are made to the database, and if it is an external database this causes major processing time issues. The training data system takes edits from the trainer and aggregates them into batch database edits. This reduces the load on the external databases greatly. At the time of writing there is a partial version of one of these in place, its development was stopped due to time constraints and the use of a memory based database.

As with all of this program, the idea is to make everything highly modular. A training method is outlined in an abstract class, as is a P value algorithm so that all may have new versions written with minimal impact of the rest of the application.

3.4 The Analysis Package

The third package is the analysis system. This consists of an analyzer and a email aggregated value algorithm. At the time of writing, there was a single analyzer and a Chi^2 aggregator algorithm.

3.4.1 The Analyzer

This is a relatively simple class. All it does is take all non-user classified messages, get an aggregated value from them, and classify them based on a threshold value.

3.4.2 The Aggregator

As explained above in background, there are many methods for aggregating word probabilities into a value for the whole message. At the time of writing, a Chi^2 module was complete. It takes a message and creates two probabilities; one for that it is relevant, one for that it is not. These values are combined and returned as outlined above by Fisher.

3.5 Putting it Together

After each package was complete, all functions were tested by comparing their effect to their expected effects. For sections involving algorithms, the algorithm was done by hand and the value compared to that which the method returned. Unfortunately, system wide testing encountered problems. During the time available for testing, the only test data available was a corpus of 2000 Enron emails. These emails were half from one sender, and half from another, also, the only tokenization that had been done was basic word extraction.

4 End Matter

While a theoretically sound filter was completed, with modules that had been tested, a full system test failed. The testing corpus was too limited, feature lacking, and writing style

specific to draw any major conclusions. Further testing will be done as soon as a better set of test data, and a better feature extraction system is available. These problems were due exclusively to the small time frame that was allowed for development, and this paper will be updated as further results are found. Further development will include new training modules and possibly new or revised database modules.

References

[1] Paul Graham, "A Plan For Spam", *http://www.paulgraham.com/spam.html*, August 2002.

[2] Jonathan Zdziarski, "Ending Spam", 2005.

[3] William S. Yerazunis, PhD, "The Spam-Filtering Accuracy Plateau at 99.9% Accuracy and How to Get Past It", *http://crm114.sourceforge.net/Plateau_Paper.pdf*, January 18, 2004.

[4] Mehran Sahami, Susan Dumaisy, David Heckermany, Eric Horvitzy , "A Bayesian Approach to Filtering Junk E-Mail", *ftp://ftp.research.microsoft.com/pub/ejh/junkfilter.pdf*

5 Acknowledgements

Kwong Yung- for giving an opportunity to work at IT.com
Jason Pratt- for working with me as a mentor on this project, and for the original ideas for this.