A Comparison of Statistical Spam Detection Techniques

By

Kevin Alan Brown

Bachelor of Science

Southwestern Oklahoma State University

Weatherford, Oklahoma

2003

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
Master of Science
May, 2006

A Comparison of Statistical Spam Detection Techniques

Thesis Approved:

J. P. Chandler

Thesis Advisor

B. E. Mayfield

I. Jonyer

A. Gordon Emslie

Dean of the Graduate College

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Email has rapidly become a common tool in everyday life. Whether it is a simple conversation or important business matter, email is an inexpensive and fast method of communication. Unfortunately, this popularity and ease of use has made email an ideal candidate for commercial marketing campaigns and scams. Users often find their inbox full of spam – unsolicited and undesirable email. What was once just an annoyance has become an epidemic for millions of email users. Tools to filter spam from legitimate email (ham) have become a necessity.

The flow of control of a typical spam filter is shown in Figure 1.1 As each email arrives, the filter

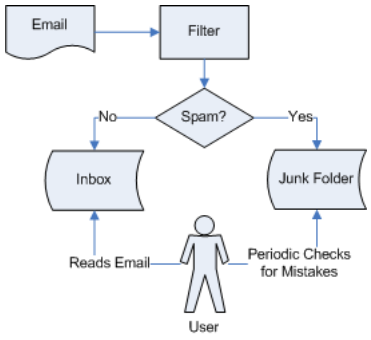

Figure 1.1: Flowchart of a Typical Spam Filter

makes its best judgment whether or not it is spam. If a message is classified as spam, it is routed to a junk folder. All ham is moved directly to the user's inbox.

Filters make two types of errors. False negatives are spam messages that are incorrectly passed to the inbox. False positives are ham messages that have been incorrectly classified as spam and sent to the junk folder. If a spam filter is noticeably effective, users can tolerate a few remaining spam in their inbox. However, all ham has a certain value to each user. If a single ham is misplaced

or even just delayed, users are negatively affected. Spam filters strive to keep the false positive rate as low as possible. No filter is perfect though, so periodic checks of the junk folder for mistakes are recommended.

All email has a header and an optional body. The header starts the message and includes important information such as the *To*, *From*, and *Date* fields. Header lines beginning with an *X-* are optional.

Spam is usually very easy to filter with the human eye. A quick glance at the *Subject* or *From* fields of a message gives a very good indicator of its spamminess. Figure 1.2 shows the contents of an actual spam. Unless the user is interested in buying watches, the *Subject* line would cause most

```
Return-Path: <wdvrwz@yahoo.com>
X-Original-To: brownba@cs.okstate.edu
Delivered-To: brownba@cs.okstate.edu
Received: from udp072122uds.hawaiiantel.net (udp072122uds.hawaiiantel.net [72.234.135.94])
  by a.cs.okstate.edu (Postfix) with SMTP id 7F405A022E
  for <brownba@cs.okstate.edu>; Wed,  1 Mar 2006 11:41:09 -0600 (CST)
Received: from sxjt.com
  by n0 (4.23.8/0.73.9) id ez114PoT83150  with SMTP; Wed, 01 Mar 2006 22:39:08 +0500
Message-ID: <20041101163013.ED8DA244AE@mailhost10.lists.techtarget.com>
Date: Wed, 01 Mar 2006 20:41:08 +0300
From: "Maurice Serrano" <wdvrwz@yahoo.com>
To: brownba@cs.okstate.edu
Subject: Cartier Watches


EXACT COPIES OF SWISS WATCHES

- exact copies of V.I.P. watches
- perfect as a gift for your colleagues and friends
- free gift box

Rolex, Patek Philippe, Omega
Cartier, Bvlgari, Franck Muller

. and 15 other most famous manufacturers.
http://ZGFiOTY1ZWZmMzYwNGQ0YTNiOTE4Zjhk.girlzboyzallluvtuna.com

 watches start from  only $180
Web Special Discountz -25%
http://ZGFiOTY1ZWZmMzYwNGQ0YTNiOTE4Zjhk.girlzboyzallluvtuna.com
```

Figure 1.2: Example of Spam

people to immediately recognize this message as spam.

Just as filters have gained in popularity and success, spammers have also improved. Spam has evolved and continues to evolve as spammers try to elude filters. Spammers may forge the *Received* and *From* lines in an attempt to appear hammy. Since they are not typically visible to users, spammers might also fill *X-* headers with hammy material. As shown below, spammy words are

often intentionally misspelled.

Vliaagra	\/IIGRA	Cialhis	ci-iallis

Again, the human eye easily recognizes the intent and knows these tokens are spammy. To a computer these tokens might be nothing more than nonsense.

There are a few types of common spam filters. Blacklisting is the practice of blocking all mail from certain servers. This can prohibit many legitimate users from getting their messages out, and spammers can easily change servers to get around the blacklists. A heuristic filter relies on human-made rules. These rules define known spam characteristics and give them weights. This paper focuses on statistical spam filters. In this method, the filter only needs pre-classified training sets of ham and spam. By giving the filter many examples of ham and spam, an original definition of spam is indirectly given. Statistical spam filters do not require hard-coded weights and rules like the heuristic approach. Instead, the example ham and spam sets give the filter a basis on which it can automatically learn acceptable classification practices.

Many of the current statistical spam filters today drew their inspiration from one web-based essay [6]. Chapter 2 looks at that essay and other important works. Chapter 3 presents the structure of a generic spam filter designed to test existing techniques. Also, a new method is introduced. Chapter 4 gives test results. Finally, Chapter 5 summarizes the results, draws conclusions, and proposes future work.

Chapter 2

Literature Review

2.1 Graham's Plans

He was not the first, but Paul Graham is widely considered to have written the seminal work on statistical spam detection. In August 2002 he posted an essay to his website titled 'A Plan for Spam' [6]. He clearly laid out an algorithm for filtering ham and spam.

The user starts with two corpora (collections of messages): one of ham, the other of spam. The initial training stage takes place first.

1. Tokenize every message.
2. Count the number of times each token appears in each corpus. Two tables are created, one for each corpus. The tables map tokens to their counts.
3. Create a third table mapping each token to its spamminess probability.

In most current spam filters, just one token database is built. It contains three columns: the token, the count of each token in the ham corpus, and the count of each token in the spam corpus. The individual token probabilities can be calculated as needed, which eliminates the need for the third table.

The first step, tokenization, is a key area of research. In his first essay, Graham used a simple definition of a token. He included alphanumeric characters, dashes, apostrophes, and dollar signs in tokens. Everything else was considered a token separator. All-digit tokens and HTML comments were ignored. Case is also ignored. Some examples of Graham tokens are listed below.

people's	$75	pills	Pharxmaceutical	Ci-iallis

Individual token probabilities are calculated by his original Lisp code in Figure 2.1, where *good* and *bad* are the token count tables produced in step 2, and *nbad* and *ngood* are respectively the number of bad and good messages. A simpler version of Graham's formula is shown in Figure 2.2.

```
(let ((g (* 2 (or (gethash word good) 0)))
      (b (or (gethash word bad) 0)))
   (unless (< (+ g b) 5)
     (max .01
          (min .99 (float (/ (min 1 (/ b nbad))
                             (+ (min 1 (/ g ngood))
                             (min 1 (/ b nbad)))))))))
```

Figure 2.1: Graham's Token Probability Function - Lisp

Graham doubled the 'good' count of a token to favor fewer false positives (ham incorrectly classified

$$
\begin{aligned}
g(w) &= \frac{2 * numTimesSeenInHam}{numHam} \\
b(w) &= \frac{numTimesSeenInSpam}{numSpam} \\
p(w) &= \frac{b(w)}{b(w) + g(w)}
\end{aligned}
$$

Figure 2.2: Graham's Token Probability Function - Simplified

as spam). Tokens are only considered if seen more than five times in total. Graham handled tokens that occur in one corpus but not the other by assigning them 0.01 or 0.99 for only ham or spam, respectively. These two values are also hard limits for token probabilities. Tokens should never be ≤ 0.0 or ≥ 1.0.

Once initial training is complete, new messages can be processed.

1. Tokenize the new message.
2. Choose the 15 unique most interesting tokens.
3. Calculate the combined probability.

Interesting tokens are those tokens farthest from a probability of 0.5 in either direction. These interesting tokens form the *decision matrix* of the filter. Graham did not say how he broke ties when filling the decision matrix. He dealt with *hapaxes* (words never seen before) by assigning them a value of 0.4, which is slightly hammy. Note, however, that tokens are still only considered if seen more than five times in total.

Graham's combined probability code is shown in Figure 2.3, where *probs* is the list of 15 interesting token probabilities. A value from 0.0 to 1.0 is returned. If the probability is more than 0.9,

```
(let ((prod (apply #'* probs)))
  (/ prod (+ prod (apply #'* (mapcar #'(lambda (x)
                                         (- 1 x))
                                     probs)))))
```

Figure 2.3: Graham's Combined Probability Function - Lisp

the message is classified as spam. A simplified version is shown in Figure 2.4. Notice a potential

$$P = \frac{x_1 x_2 \dots x_{15}}{x_1 x_2 \dots x_{15} + (1-x_1)(1-x_2)\dots(1-x_{15})}$$

Figure 2.4: Graham's Combined Probability Function - Simplified

problem if hard limits were not used. If two tokens had probabilities of 0.0 and 1.0, a divide-by-zero error would occur.

Graham refers to his method as Bayesian filtering [11]. However, the term Bayesian filtering is now used as a catch-all phrase for statistical spam filters loosely based on Graham's work. Bayes' rule is shown in Figure 2.5. In the context of spam filtering, C is the condition that 'the message is spam',

$$P(C|F) = \frac{P(F|C)P(C)}{P(F|C)P(C) + P(F|C')P(C')}$$

Figure 2.5: Bayes' Rule

C' means 'the message is not spam', and F is the feature being considered (the token). $P(C|F)$ is the probability a message containing the feature is spam. This the desired overall probability, P, we are after. $P(F|C)$ is the probability a spam message contains the feature. This is represented by the individual token probabilty, $p(w)$, in Figure 2.2. $P(C)$ is the probability a random message is spam. Graham's combined probability equation, shown in Figure 2.6, simplifies Bayes' rule. Substituting

$$P(C|F) = \frac{P(F|C)}{P(F|C) + P(F|C')}$$

Figure 2.6: Graham's Bayes' Rule

x for $P(F|C)$ and $(1-x)$ for $P(F|C')$, and accounting for many features, gives Graham's combined

probability function in Figure 2.4. This corresponds to assuming $P(C) = P(C') = 0.5$, equal a priori probabilities that a message is spam or ham.

Graham's method results in probabilities with little uncertainty. Most message classification scores end up close to either 0.0 or 1.0. Consider the decision matrices in Table 2.1. Examples 1

	Ex1	Ex2	Ex3	Ex4
Token Probabilities	0.01	0.99	0.99	0.99
	0.01	0.99	0.99	0.99
	0.01	0.99	0.99	0.99
	0.01	0.99	0.99	0.99
	0.01	0.99	0.99	0.99
	0.01	0.99	0.99	0.99
	0.01	0.99	0.99	0.99
	0.01	0.99	0.01	0.99
	0.01	0.99	0.01	0.01
	0.01	0.99	0.01	0.01
	0.01	0.99	0.01	0.01
	0.01	0.99	0.01	0.01
	0.01	0.99	0.01	0.01
	0.01	0.99	0.01	0.01
	0.01	0.99	0.01	0.01
Combined Probability	0.000000	1.000000	0.010000	0.990000

Table 2.1: Graham's Combined Probabilities

and 2 behave as expected. If only hammy or spammy tokens are used, the combined probability is confidently hammy or spammy, respectively. However, notice the scores of examples 3 and 4. In example 3, hammy tokens have the majority with eight of the fifteen tokens. The remaining seven tokens are spammy, but the combined probability is a very confident 0.01. A similar behavior is shown in example 4. Once spammy tokens take the majority, the combined probability flips to 0.99. This radical change in the combined probability due to a change in only one position in the table is unreasonable, as has been pointed out by later researchers [2][13].

A year after his first plan, Paul Graham wrote an update to 'A Plan for Spam', titled 'Better Bayesian Filtering' [7]. He presented a more elaborate definition of a token. Now he suggested preserving case. Previously, periods and commas were treated as delimiters, but they are now included in tokens if they are between two digits. This approach allows IP addresses and prices to remain intact.

Graham's better plan also included the idea of marking header data. Tokens within specific header fields were marked as such. For example, if the token *brownba@cs.okstate.edu* is found in the *To* field of a header, that token would become *To*brownba@cs.okstate.edu* (where * is some

character not allowed in tokens). At the time, Graham marked tokens inside the *To*, *From*, *Subject*, and *Return-Path* lines, and within URLs. Graham also discussed what to do about HTML. He settled on noticing some tokens and ignoring the rest. He focused on the *a*, *img*, and *font* tags in HTML, as these are likely to contain URLs.

In 'Better Bayesian Filtering', Paul Graham also presented a more theoretical topic of degeneration. Marking header tokens and including more types of tokens will increase the filter's vocabulary. This can make a filter more discriminating, but with a growing vocabulary, the probability that a token has never been seen before also rises. Degeneration allows a new token to be treated as a less specific version of itself. The premise is that a new token's probability of 0.4 is probably not as accurate and useful as the probability of some similar token seen already. For example, if the token *Subject*longer!!!* is not found in the database, the following degenerate case would be tried: *Subject*longer*, *Subject*Longer!!!*, *Subject*longer*, *longer!!!*, *Longer!!!*, *longer*, etc. The probability of the degenerate case farthest from 0.5 would be used. This token's probability would most likely be more indicative than 0.4.

Paul Graham's personal filter is effective. He trained his filter with ham and spam corpora each of about 4000 messages. Over the next year, he received about 1750 spam. He claims to have caught 99.5% of spam with 0.03% false positives over that period.

2.2 Pantel and Lin

The AAAI-98 Workshop on Learning for Text Classification took place four years before Graham's first essay on spam detection. Two papers presented at this conference, one by Pantel and Lin [12] and the other by Sahami, Dumais, Heckerman, and Horvitz of Microsoft Research [15], formed the foundation for our current state-of-the-art spam filters.

Catching 92% of spam with 1.16% false positives, Pantel and Lin's filter performed better than the filter from Microsoft Research. However, this is noticeably worse than Paul Graham's 99.5%/0.03% accuracy achieved four years later. A few differences in the way Pantel and Lin operated compared to Graham, outlined below, could have attributed to the decreased accuracy.

The first difference is the data Pantel and Lin used. They used what is considered a very small set of training messages: 160 spam and 466 ham. In contrast, Graham trained with about 4000 messages each of spam and ham. With such a small training set as that used by Pantel and Lin, many tokens in the testing phase would be new and thus considered slightly hammy. Also, not only did they train with few messages, their messages were not complete. They removed the headers from

all messages. With the classification based solely on the body of the message, a lot of potentially incriminating data has been lost. It is highly recommended not to remove any information from your messages.

The data fed into Pantel and Lin's filter was substantially different from Graham's data, and so was the way they tokenized. They defined a token in two ways. A token may be a consecutive sequence of letters or digits, or it can be a consecutive sequence of non-space, non-letter, and non-digit characters. Tokens of the second type are limited to a maximum length of three characters. Additionally, Pantel and Lin used an algorithm to remove suffixes from tokens. For example, the token *waited* would be reduced to *wait*, and *meetings* would be treated as *meet*. This 'stemming' could have been an optimization or a step to combat the small set of training data. Examples of tokens in Pantel and Lin's vocabulary are shown below.

***	$	99999	you	address	stem

Pantel and Lin used another interesting technique to derive information from their data. Instead of stripping suffixes, they pulled trigrams from words. They defined a trigram as each three letter sequence of consecutive letters in a word. A large amount of information is lost when words are reduced to trigrams. However, this reduction did not significantly hurt their performance.

Pantel and Lin, and Sahami et al. deserve the credit for originating the idea of a statistical spam filter, although similar techniques had been used for decision processes in other contexts. Paul Graham made the process more efficient and more widely known.

2.3 SpamProbe

SpamProbe is an open-source spam filter developed by Brian Burton [2]. Burton credits Paul Graham for the initial ideas, but Burton has implemented some alternative approaches designed to improve performance.

SpamProbe's tokenizer boasts more rules than those originally proposed by Graham. Some example SpamProbe tokens are shown below.

127.0.0.1	$10,000	Hto_undisclosed	cs.okstate.edu	ci-iallis

The tokenizer allows certain non-text characters ('.', ',', '+', '-', '_', and '$') within tokens. All other non-alphanumeric characters are delimiters. Purely numeric tokens are ignored. The token *127.0.0.1* is valid, but *127* is not. All tokens are converted to lower case, which will lead to a smaller database. Tokens containing punctuation are broken down by repeatedly removing the head of the token. For

example, *cs.okstate.edu* will result in tokens *cs.okstate.edu*, *cs*, *okstate.edu*, *okstate*, and *edu*. This is designed to capture domain names from URLs. Graham's individual token probability function is retained, but the hard limits are now 0.000001 and 0.999999, and the hapax value is 0.300000.

SpamProbe has many user-configurable options. For example, it can recognize HTML tags, but by default ignores them. In either case, whether all or no HTML tags are used, URLs inside HTML are always retained. By default, header data is marked for tokens inside the *Received*, *Subject*, *To*, *From*, and *Cc* lines. This is referred to as the 'normal' set of header fields. The marked set can be changed to all header fields, no header fields, or all header fields excluding *X-* fields. The *X-* header fields in any email consist of optional lines added by user email clients. Spammers have been known to insert seemingly hammy material in *X-* header fields, since these fields are not usually visible to users. For example, *X-mailer* is a common *X-* header line. Spammers can insert the name of a common email client to give the illusion that messages were sent from that client. Header tokens are marked by prefixes consisting of an *H*, the field name, and an '_'. For example, if the term *tok* was in the *To* field, the token *Hto_tok* would be produced. Since SpamProbe converts all terms to lower case, marked header tokens will never be confused with body tokens.

In his first plan, Paul Graham mentioned the idea of tokenizing word pairs instead of just single words. Burton has implemented this idea in SpamProbe. By default, all single and two-word phrases are counted. For example, when the string 'one two three' is tokenized, the tokens 'one', 'one two', 'two', 'two three', and 'three' are generated. Optionally, the user can choose any phrase length. This idea of word pairs gives the tokenizer a sense of context.

An important difference between SpamProbe and Graham's filter is the decision matrix. Graham used the fifteen most interesting, unique tokens in every case. Burton implemented a more dynamic approach in SpamProbe. By default, a decision matrix of 27 tokens is used. Furthermore, tokens may be repeated up to two times if they appear in the message twice. Both the window size and the number of repeats may be adjusted by the user. A potentially important note should be made regarding tokens that have never been seen before. SpamProbe scores these tokens with a constant value like Graham, but they are allowed to appear in the decision matrix if slots remain empty. In other words, SpamProbe will fill all slots of a decision matrix if the message size is greater than or equal to the size of the decision matrix.

Optionally, a variable-sized array of tokens can be used in SpamProbe. This array starts at size five and allows tokens to repeat up to five times each. To prevent a single token from dominating the window, the array size is variable. All significant tokens of probability ≤ 0.1 or ≥ 0.9 in the message are added to the array. Burton claims slightly lower spam detection accuracy but fewer

false positives with this approach.

Brian Burton also addressed the lack of uncertainty in Graham's combined probability function. SpamProbe uses the modified function shown in Figure 2.7. This small change of using the n^{th} root

$$\begin{aligned} S &= (x_1 x_2 \dots x_n)^{1/n} \\ G &= ((1-x_1)(1-x_2)\dots(1-x_n))^{1/n} \\ P &= \frac{S}{S+G} \end{aligned}$$

Figure 2.7: SpamProbe's Combined Probability Function

of products produces smoother probabilities. As seen in Table 2.2, examples 1 and 2 still perform

	Ex1	Ex2	Ex3	Ex4
Token Probabilities	0.01	0.99	0.99	0.99
	0.01	0.99	0.99	0.99
	0.01	0.99	0.99	0.99
	0.01	0.99	0.99	0.99
	0.01	0.99	0.99	0.99
	0.01	0.99	0.99	0.99
	0.01	0.99	0.99	0.99
	0.01	0.99	0.01	0.99
	0.01	0.99	0.01	0.01
	0.01	0.99	0.01	0.01
	0.01	0.99	0.01	0.01
	0.01	0.99	0.01	0.01
	0.01	0.99	0.01	0.01
	0.01	0.99	0.01	0.01
	0.01	0.99	0.01	0.01
Combined Probability	0.010000	0.990000	0.424008	0.575992

Table 2.2: SpamProbe's Combined Probabilities

similarly to Graham's function. However, now examples 3 and 4 give much more meaningful values. Burton also differs from Graham in using a 0.7 spam threshold.

Burton claims over 99% accuracy using SpamProbe with his own email. However, accuracy claimed by authors and researchers should not be expected by all users. Everybody's email is different, and often corpora show a plateau that is rarely surpassed with any filter optimization.

2.4 Gary Robinson

The development of two additional combination functions is credited to Gary Robinson [13]. These functions have been employed with great success in many spam filters.

Robinson's geometric mean function is shown in Figure 2.8. This function is quite similar to

$$\begin{aligned} P &= 1 - \sqrt[n]{((1-p_1)*(1-p_2)*\ldots*(1-p_n))} \\ Q &= 1 - \sqrt[n]{(p_1*p_2*\ldots*p_n)} \\ S &= \frac{1+\frac{(P-Q)}{(P+Q)}}{2} \end{aligned}$$

Figure 2.8: Robinson's Geometric Mean Function

Burton's combination function in SpamProbe. They both use the n^{th} root of products and return values other than 0.0 or 1.0.

Robinson has also proposed an altered token probability function [14]. He has named this function $f(w)$, in Figure 2.9, a degree of belief. In this function, $p(w)$ can be calculated as before in Graham's

$$f(w) = \frac{(s*x)+(x*p(w))}{s+n}$$

Figure 2.9: Robinson's Degree of Belief Function

essay, s is a tunable constant, x is an assumed probability given to words never seen before (hapaxes), and n is the number of messages containing this token. Initial values of 1 and 0.5 for s and x, respectively, are recommended. Robinson suggests using this function in situations where the token has been seen just a few times. An extreme case is where a token has never been seen before. In this case, the value of x will be returned. As the number of occurrences increases, so does the degree of belief.

In Robinson's degree of belief function, $p(w)$ can be calculated as Graham did, but he suggests another slight modification [14]. Figure 2.10 shows how instead of using the total number of occurrences of a token in a ham or spam corpus, Robinson used the number of messages containing that token. Robinson believes Graham's method performs slightly better than his since Graham's

$$\begin{aligned} g(w) &= \frac{numHamWithToken}{numHam} \\ b(w) &= \frac{numSpamWithToken}{numSpam} \\ p(w) &= \frac{b(w)}{b(w)+g(w)} \end{aligned}$$

Figure 2.10: Robinson's Token Probability Function

counting method does not ignore any of the token occurrences data.

The second combining function Robinson has proposed is based on the work of Sir Ronald Fisher. This method has been named the Fisher-Robinson Inverse Chi-Square Function [14]. There are three parts to this equation, as shown in Figure 2.11. H is the combined probability sensitive

$$
\begin{aligned}
H &= C^{-1}\Big(-2\ln\prod_{w} f(w), 2n\Big) \\
S &= C^{-1}\Big(-2\ln\prod_{w}\Big(1-f(w)\Big), 2n\Big) \\
I &= \frac{H}{H+S}
\end{aligned}
$$

Figure 2.11: Fisher-Robinson's Inverse Chi-Square Function

to hammy values, S calculates the probability sensitive to spammy values, I is used to produce the final probability in the usual 0 to 1 range, C^{-1} is the inverse chi-square function, and n is the number of tokens used in the decision matrix. Jonathan Zdziarski [21] gives the C code for C^{-1} in Figure 2.12. Zdziarski notes the high level of uncertainty provided by this function. SpamBayes is

```
double chi2Q( double x, int v )
{
  int i;
  double m, s, t;

  m = x / 2.0;
  s = exp( -m );
  t = s;

  for( i=1; i<(v/2); i++ ){
    t *= m / i;
    s += t;
  }
  return (s < 1.0) ? s : 1.0;
}
```

Figure 2.12: The Inverse Chi-Square Function: C^{-1}

a free and open-source spam filter that uses the Fisher-Robinson Inverse Chi-Square Function [17]. The uncertainty given by this function allows SpamBayes to return an Unsure result instead of just Ham or Spam. SpamBayes is also noted for using a slightly different function for I, where $I = \frac{1+H-S}{2}$.

Chapter 3

A Spam Detection Test System

3.1 System Overview

Statistical spam filters have a few common modules. However, the specifics of how these modules work can vary greatly. Tokenizers can be very simple or extremely elaborate. The combination function might be a direct implementation of Graham's function, or something original and possibly proprietary. To compare the effect of different techniques, I designed and implemented a spam detection test system (known as the *System* from here on). A flowchart of the System is shown in Figure 3.1. The System, written in C++, implements existing approaches and a few proposed ideas.

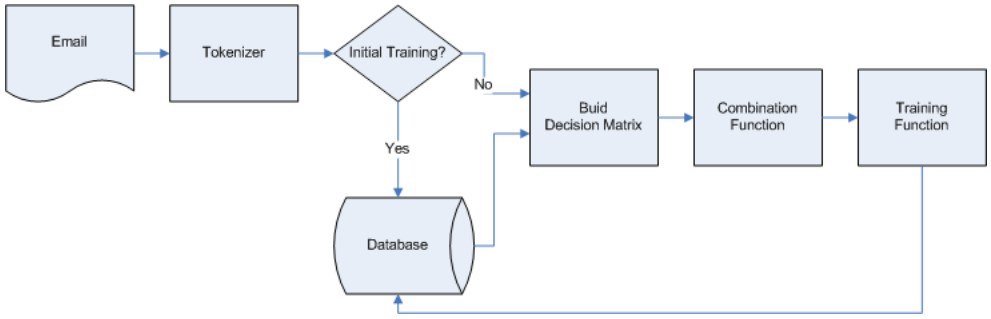

Figure 3.1: Flowchart of Proposed System

3.2 Tokenizer

The tokenizer can be thought of as the eyes of the filter. It determines what data is pulled from a given message. Current spam filters have employed a variety of tricks to try to gain as much

knowledge as possible from each message. Many questions of how to handle certain parameters remain. For simplicity, I used tokenization code from the open-source SpamProbe project.[1]

The method of marking header data that Graham presented is commonly believed to be a good one. One advantage is that strong tokens (those whose probability is far from 0.5 in either direction) could appear more often in the decision matrix. For example, if the spammy token *tok* is found in both the body and *To* field, both *tok* and *Hto_tok* could appear in the decision matrix and influence the overall probability. The SpamProbe model of marking header data, given in Section 2.3, is used. A test will be conducted to determine the effectiveness of marking header tokens. In addition, the effects of tokenizing just subsets of the headers will also be compared. The tokenizing of all headers, a 'normal' set of headers *(From, To, Cc, Subject, and Received)*, and all header fields except *X-* lines will be compared.

Word phrases will also be tested. The technique of tokenizing pairs of words was initially proposed by Graham and has been implemented in many popular spam filters. The tokenizing of pairs and triples will be tested against single word tokens. When n-word phrases are used, all phrases less than n are also included. Much like marking header data, word pairs gives tokens a sense of context and situation. Consider the tokens in Table 3.1. The tokens and counts are actual values from the

HamCount	SpamCount	Token
396	500	number
293	360	order
70	77	sending
15	0	order number
0	20	order sending

Table 3.1: Pairs vs Singles

X corpus described later. Singly, *number*, *order*, and *sending* appear fairly neutral. Using pairs tells a different story, as together these tokens can appear completely hammy or spammy.

3.3 Weighting

In an effort to determine how effective marking header data and using phrased tokens are, the benefits of weighting header data and phrased tokens higher (or lower) than their body and single-word counterparts will be tested. To test this idea, a new token probability function was developed together with John P. Chandler [3]. It is shown in Figure 3.2. The *headerWeight* and *phraseWeight* are defaulted to 1.0, meaning they have no effect. Each weight can be set to a value w, where

[1] See Appendix A for details

$$
\begin{aligned}
weight &= headerWeight * phraseWeight \\
g(w) &= \frac{weight * numTimesSeenInHam + eps}{numHam + eps} \\
b(w) &= \frac{weight * numTimesSeenInSpam + eps}{numSpam + eps} \\
p(w) &= \frac{b(w)}{b(w) + g(w)}
\end{aligned}
$$

Figure 3.2: Weighted Token Probability Function

$w > 0.0$. If $0.0 < w < 1.0$, the token is weighted lower. For example, a spammy token's probability would move closer to 0.5. Likewise, if $w > 1.0$, the token is weighted higher. A spammy token's probability would then be pushed farther towards 1.0. This action effectively changes the confidence of token probabilities. The farther a probability is from 0.5 in either direction, the more likely it is to be chosen for the decision matrix, where it will impact the overall combined probability. The variable *eps* is a constant tuned for performance. Using the variable *eps* also has the side effect of not requiring hard limits on any token probabilities. Graham gave ham-only and spam-only tokens values of 0.01 and 0.99, respectively. Now with an *eps* value not equal to zero, neither $g(w)$ nor $b(w)$ will equal zero, and hard limits will not be necessary. Without hard limits, $g(w)$, $b(w)$, and $p(w)$ are now smooth functions, which is more favorable for possible optimization techniques.

Weighting header fields and phrase tokens will be tested separately. When header weighting is applied, header tokens, regardless of whether or not they are single or two-word tokens, are given the specified weight. As explained above, this weight strengthens or weakens the individual probability of those tokens. The remaining tokens (all body tokens) are given unit weight (1.0), meaning they are not strengthened or weakened. Phrase weighting is similar. All single-word tokens are given unit weight. All other tokens (which are of phrase size > 1), regardless of whether or not they are in a header field, are given the specified strengthening or weakening weight. Since the strengthening and weakening action has a direct effect on which tokens appear in the decision matrix, it cannot be expected that a header weight of 0.5 and a phrase weight of 1.0 would give results equal to a header weight of 1.0 and a phrase weight of 2.0. For example, when a header weight of 0.5 and a phrase weight of 1.0 is used, header tokens are weakened. The resulting decision matrix may be different than if both the header and phrase weights were 1.0, or the resulting decision matrix could contain the same tokens compared to header and phrase weights of 1.0, but the overall score would be changed due to the weakened header tokens.

In non-weighted tests, Graham's individual token probability function will be used. Like SpamProbe, hard limits of 0.000001 and 0.999999 will be used with Graham's individual token probability function.

3.4 Combination Functions

The tokenizer is responsible for pulling all possible data from each message. Each token is then given a value using an individual token probability function. It is the job of the combination function to gather these individual probabilities and make a decision. Three combination functions are implemented in the System and will be tested: Graham's original in Figure 2.4, SpamProbe's in Figure 2.7 (hereinafter known as *SP-Graham*), and Gary Robinson's geometric mean in Figure 2.8.

Vital to the performance of any combination function is the building of the decision matrix. Choosing the tokens on which the combination function bases its decision is an important step. However, there are many variables. The number of tokens and the number of repeats allowed could be tested, but for simplicity, these variables will be held constant for most tests. SpamProbe's model of 27 tokens with 2 repeats will be used primarily. The top 27 tokens are chosen from a message whose tokens have been sorted. The sort criterion is first by the token's score's distance from 0.5, then ties are broken by favoring hammy tokens.

However, this work will differ from SpamProbe in the handling of new tokens. Tokens that do not meet a constant maturity level will not be allowed in any decision matrix. Maturity is based on the total of database ham and spam counts for each token. Currently the maturity level is set to five, as Graham suggested. If the decision matrix is not full after adding all mature tokens, the combination function still functions, and a result will be returned. With this course of action, the token hapax value will never be used. In the rare situation that a decision matrix is empty, the value 0.4 (ham) will be returned as the overall score. SpamProbe differs in that the decision matrix will be filled if there are tokens to fill it, even if those tokens do not have sufficient database counts.

3.5 Training

Any spam filter will make mistakes. However, a key benefit of statistical spam filters is their ability to adapt. After a new message is scored, various methods may be employed to update (train) the token database. Three variations have been implemented and will be tested.

The first technique is to train on everything *(TEFT)*. Since it requires no human intervention,

this is also known as unsupervised learning. Every message received is scored, and its tokens are added to the database, whether the classification was correct or not. For example, if a message is classified as spam, the spam count of all tokens in that message will be incremented or added to the database with a value of one if they are new.

An alternative to *TEFT* has been implemented that employs error correction *(TEFT-Corrective)*. In a simulation, the correct classification is known, so an immediate error correction can be employed. This will be acceptable for a simulation, but is not practical in a normal situation. In a real-life situation, many subsequent classifications and database updates may have occurred before the user recognized the error and issued a correction request. A mistake is corrected by re-tokenizing the message, then decrementing the counts in the incorrect column and incrementing the counts in the correct column.

Another technique is to train only on errors *(TOE)*. Only when the filter incorrectly classifies a message will the database be updated. Again, immediate corrections will be required, which is not practical for production applications. *TOE* has the benefit of fewer database writes and should create a database of fewer tokens. However, a smaller, infrequently-updated database could hurt accuracy when dealing with new types of spam.

The initial training phase is also important to the performance of any spam filter. Paul Graham's accuracy of 99.5% was based on tests using ham and spam corpora with about 4000 messages in each. An argument could be made that this is not typical of the average user. I suspect most users do not have 4000 ham messages archived, waiting for the day when they will train a spam filter. Nor do they have 4000 spam messages waiting. Spam is junk, and is therefore usually deleted immediately when found. Tests will be performed to see how accuracy is affected by different initial training set sizes. However, most tests will be conducted with a training set size of 5000 messages (total of ham and spam).

3.6 Testing

Testing will be performed in a manner similar to the style William Yerazunis suggested [20]. For each corpus, the ham and spam will be shuffled, creating randomized index files. The index files contain the path to each message and their gold-standard (correct) classification. Five such shuffled index files per corpus will be used. In the results given, the number of messages and errors are the sums of those from the five indexes. For each index, the first n messages will be used for initial training, then the rest of the messages in that index will be classified and perhaps used also for

training. Most test configurations will use a training set size of 5000 messages. After each index is complete, the token database will be deleted to ensure an accurate test for the next index. The index files have been preserved, so each test configuration will use the same ordering of messages.

Accuracy is the most important measure of performance in spam filtering, but we are dealing with two different types of errors. The error measurements are defined in Figure 3.3 [4]. The false

$$\begin{aligned} \text{True Negatives (ham classified as ham)} &= a \\ \text{False Negatives (spam misclassified as ham)} &= b \\ \text{False Positives (ham misclassified as spam)} &= c \\ \text{True Negatives (spam classified as spam)} &= d \end{aligned}$$

$$\begin{aligned} \text{False Positive Rate} &= \frac{c}{a+c} \\ \text{False Negative Rate} &= \frac{b}{b+d} \\ \text{Overall Error Rate} &= \frac{b+d}{a+b+c+d} \\ \text{Overall Accuracy} &= \frac{a+d}{a+b+c+d} \end{aligned}$$

Figure 3.3: Error Rates Defined

positive rate is the percentage of all ham that are misclassified. The false negative rate is defined similarly. False positives are considered much worse than false negatives. Users can accept a small percentage of spam passed through to their inbox, but any ham misclassified as spam could have unfortunate consequences. Typically, a spam filter channels any email classified as spam to a junk folder. Depending on their confidence in their spam filter, users might rarely or never check this junk folder for false positives. For these reasons, I will weigh the false positive count highly when comparing two configurations. When relevant, the average number of database tokens per shuffle will be noted.

Testing will be conducted with two private email collections (X of Kevin Brown and Y of John Chandler) and with the publicly available SpamAssassin corpus (SA) [18]. Properties of the three corpora are shown in Table 3.2. The ham in X is comparatively homogeneous, consisting mainly

	X	Y	SA
Ham	2470	3550	4150
Spam	5368	6825	1891

Table 3.2: Corpora Properties

of personal correspondence plus course-related messages. The number of original senders of ham in this corpus is low. The ham in Y also contains significant numbers of commercial ads and purchases, medical email messages, mail from students in two courses, and mail received as graduate coordinator of a department in a large university. Therefore, the messages in corpus Y are quite heterogeneous and are expected to be harder to classify correctly than the messages in corpus X.

Testing will be done in a safe environment where all known viruses have been removed from the corpora. Three corpora are used for testing because everybody's email is different. Some corpora are inherently easy to classify, while others are not as cooperative. I am looking for solutions that benefit all types of users, so a filter configuration that succeeds on just one corpus cannot receive a full recommendation if the other corpora exhibit decreased performance.

Chapter 4

Results

4.1 Standard Configurations

First, the base configuration is presented and tested against a configuration similar to the original model Graham proposed. This base setup is similar to the default options supported by SpamProbe. One difference is that all header lines are tokenized and marked, whereas, by default, SpamProbe only utilizes the 'normal' set of header lines (*Received*, *Subject*, *To*, *From*, and *Cc*). This base setup was used as a starting point in many of the following tests. Table 4.1 lists the options for the *Base* and *Graham-like tests*. Graham's original model did not mark header data, and it used just single-

Option	Base	Graham-like
Initial Training Set Size	5000	5000
Decision Threshold	0.7	0.7
Post-Classification Training Mode	TEFT-Corrective	TEFT-Corrective
New Word Probability	0.4	0.4
Token Probability Function	Graham	Graham
Combined Probability Function	SP-Graham	Graham
Marked Header Lines	All	None
Maximum Phrase Length	2	1
Decision Matrix Size	27	15
Token Repeats in Matrix	2	1
Graham-like Double Ham Count	False	True

Table 4.1: Base and Graham-like Configurations

word tokens. Its decision matrix is smaller than the default model of SpamProbe. However, the decision matrix of SpamProbe does allow each token to fill two slots (if that token appears twice in the message), so a minimum of fourteen unique tokens are needed. As seen in Table 4.2, despite all the differences, these two configurations gave similar results. A possible cause for concern is in the Y

Corpus		Configuration Base	Graham-like
X	Overall Accuracy	0.997674	0.998450
	False Positive Rate	0.003197	0.003197
	False Negative Rate	0.001937	0.000815
	Ham Messages	4379	4379
	False Positives	14	14
	Spam Messages	9811	9811
	False Negatives	19	8
	Avg DB Token Count	925993	225921
Y	Overall Accuracy	0.957730	0.943107
	False Positive Rate	0.000000	0.000220
	False Negative Rate	0.063917	0.085917
	Ham Messages	9102	9102
	False Positives	0	2
	Spam Messages	17773	17773
	False Negatives	1136	1527
	Avg DB Token Count	1449288	295528
SA	Overall Accuracy	0.974063	0.967723
	False Positive Rate	0.000848	0.000000
	False Negative Rate	0.079089	0.100659
	Ham Messages	3536	3536
	False Positives	3	0
	Spam Messages	1669	1669
	False Negatives	132	168
	Avg DB Token Count	915982	189235

Table 4.2: Base and Graham-like Results

corpus where false positives appeared with the *Graham-like* test, and false negatives were noticeably higher in *Y* and *SA*. False positives are always a concern, and here there is an inconclusive trend regarding them. The *Y* corpus had two false positives under the *Graham-like* setup, and the *SA* corpus had three under the *Base* setup. Each of these two corpora had zero false positives with the other setup. An obvious result is the substantially smaller database with the *Graham-like* setup, due to the lack of marking header data and the maximum phrase length of one.

4.2 Training Modes and Initial Training Set Sizes

With a production software product like SpamProbe it is not atypical to see a user's token database consume over 40 megabytes of disk space. On a modern desktop computer where hard drives over 100 gigabytes are common, this amount of storage is very reasonable. However, in a multi-user server environment where each user is granted a small amount of disk space, 40 megabytes could be too much to justify. For example, if individual users are each granted just 100 megabytes of storage, to use almost half that amount just for spam detection is hard to defend.

Production spam filters employ techniques to limit database growth. A manual cleanup operation

is commonly supported. Periodically, users purge certain tokens, such as tokens not modified for n days, from their database. I looked at a method to minimize database updates, thereby limiting growth.

With the *TOE* method described in Section 3.5, the database is only updated when the user corrects an error. Since errors are usually in a small minority, database updates should be few. *TOE* was tested against *TEFT-Corrective*. In this simulation, I assumed an ideal situation where the user notices every error and corrects each before the next message classification has begun. Additionally, results with *TEFT* (non-corrective) are included. Tests were conducted with the *Base* configuration in Table 4.1, only differing by the training mode.

While investigating these three training modes, the initial training set size was also studied. As mentioned in Section 3.5, many users might not have large corpora of ham and spam saved to build their initial database. It is worthwhile to see what impact small initial training sets have on accuracy. Results for *TEFT-Corrective*, *TOE*, and *TEFT* are shown in Tables 4.3, 4.4, and 4.5, respectively. As outlined in Section 3.6, the classification set of messages is all messages remaining after initial training. Therefore, in these tests, as the initial training set size increases, the number of classified messages decreases. Since the number of classified messages now differs between tests, the numbers of false positives and false negatives cannot be directly compared. The false positive and false negative *rates* should be compared.

		Initial Training Set Size						
Corpus		0	50	100	500	1000	2500	5000
X	Overall Accuracy	0.993595	0.994453	0.994701	0.995476	0.996285	0.997152	0.997674
	False Positive Rate	0.013036	0.011428	0.010762	0.008505	0.006007	0.003766	0.003197
	False Negative Rate	0.003353	0.002848	0.002791	0.002702	0.002677	0.002438	0.001937
	Ham Messages	12350	12251	12173	11523	10655	8231	4379
	False Positives	161	140	131	98	64	31	14
	Spam Messages	26840	26689	26517	25167	23535	18459	9811
	False Negatives	90	76	74	68	63	45	19
	Avg DB Token Count	925993	925993	925993	925993	925993	925993	925993
Y	Overall Accuracy	0.929889	0.931351	0.932380	0.938552	0.942997	0.951162	0.957730
	False Positive Rate	0.000901	0.000340	0.000342	0.000297	0.000252	0.000075	0.000000
	False Negative Rate	0.106110	0.104154	0.102585	0.093048	0.086101	0.073660	0.063917
	Ham Messages	17750	17656	17569	16822	15888	13282	9102
	False Positives	16	6	6	5	4	1	0
	Spam Messages	34125	33969	33806	32553	30987	26093	17773
	False Negatives	3621	3538	3468	3029	2668	1922	1136
	Avg DB Token Count	1449288	1449288	1449288	1449288	1449288	1449288	1449288
SA	Overall Accuracy	0.963218	0.965048	0.965932	0.968273	0.969530	0.970912	0.974063
	False Positive Rate	0.001783	0.001312	0.001275	0.001211	0.001159	0.001078	0.000848
	False Negative Rate	0.113591	0.108730	0.105862	0.098221	0.094076	0.088991	0.079089
	Ham Messages	20750	20574	20391	18990	17254	12064	3536
	False Positives	37	27	26	23	20	13	3
	Spam Messages	9455	9381	9314	8715	7951	5641	1669
	False Negatives	1074	1020	986	856	748	502	132
	Avg DB Token Count	915982	915982	915982	915982	915982	915982	915982

Table 4.3: TEFT-Corrective Tests

With *TEFT-Corrective*, the database is always updated, and updated correctly. Not surprisingly, all corpora showed improved overall accuracy as the initial training set grows. Both the false positive rate and the false negative rate dropped in all but one test. Even with zero initial training, all three corpora presented respectable accuracy. In fact, increasing the training set from zero to 5000 messages only increased the overall accuracy of corpus *X* by 0.4079%. Its false positive rate started at 1.3% and dropped to just 0.3197%. Corpus *SA* behaved much like *X*. Corpus *Y* showed less than a 3% reduction of overall accuracy with no initial training compared to a training set of 5000 messages. However, at 5000 this corpus did not give any false positives. Still, with no initial training, its false positive rate of 0.0901% was very reasonable.

		Initial Training Set Size						
Corpus		0	50	100	500	1000	2500	5000
X	Overall Accuracy	0.970350	0.980945	0.985552	0.992096	0.993946	0.995879	0.997322
	False Positive Rate	0.022915	0.016978	0.017005	0.015794	0.012482	0.007532	0.003654
	False Negative Rate	0.032750	0.020008	0.013275	0.004291	0.003144	0.002600	0.002242
	Ham Messages	12350	12251	12173	11523	10655	8231	4379
	False Positives	283	208	207	182	133	62	16
	Spam Messages	26840	26689	26517	25167	23535	18459	9811
	False Negatives	879	534	352	108	74	48	22
	Avg DB Token Count	58481	52342	56600	122814	199446	390586	659363
Y	Overall Accuracy	0.947933	0.953259	0.959027	0.967392	0.964928	0.962997	0.961488
	False Positive Rate	0.020620	0.016708	0.013717	0.005231	0.002392	0.000903	0.000439
	False Negative Rate	0.068425	0.062351	0.055138	0.046755	0.051828	0.055379	0.058009
	Ham Messages	17750	17656	17569	16822	15888	13282	9102
	False Positives	366	295	241	88	38	12	4
	Spam Messages	34125	33969	33806	32553	30987	26093	17773
	False Negatives	2335	2118	1864	1522	1606	1445	1031
	Avg DB Token Count	121698	126945	134222	237694	367459	617799	929897
SA	Overall Accuracy	0.960271	0.965715	0.970847	0.982891	0.981789	0.976221	0.975600
	False Positive Rate	0.034602	0.028531	0.024913	0.006372	0.002898	0.001492	0.000848
	False Negative Rate	0.050978	0.046903	0.038437	0.040505	0.051440	0.071441	0.074296
	Ham Messages	20750	20574	20391	18990	17254	12064	3536
	False Positives	718	587	508	121	50	18	3
	Spam Messages	9455	9381	9314	8715	7951	5641	1669
	False Negatives	482	440	358	353	409	403	124
	Avg DB Token Count	113154	121294	120973	193081	292678	527415	811547

Table 4.4: TOE Tests

The *TOE* method did substantially reduce the database token count. With the standard initial training set size of 5000, corpus *Y* displayed the largest reduction of tokens at over 64%. Compared to *TEFT-Corrective*, this corpus also enjoyed increased overall accuracy, although it came at the expense of more false positives. Corpus *SA* also experienced higher overall accuracy with *TOE*. Interestingly, the overall accuracy of corpora *Y* and *SA* peaked with a training set of 500, then slightly declined. Corpus *X* performed well with *TOE*, but never quite reached the level of accuracy given by *TEFT-Corrective*. The higher false positive rates make *TOE* a very questionable choice unless database size is a primary concern, in which case alternative approaches (such as limiting the

phrase length to one word) should also be considered.

Corpus		Initial Training Set Size 0	50	100	500	1000	2500	5000
X	Overall Accuracy	0.315131	0.914689	0.926338	0.982666	0.991284	0.995654	0.996899
	False Positive Rate	0.000000	0.267733	0.227388	0.044606	0.017457	0.007532	0.005481
	False Negative Rate	1.000000	0.001574	0.003092	0.004848	0.004759	0.002925	0.002039
	Ham Messages	12350	12251	12173	11523	10655	8231	4379
	False Positives	0	3280	2768	514	186	62	24
	Spam Messages	26840	26689	26517	25167	23535	18459	9811
	False Negatives	26840	42	82	122	112	54	20
	Avg DB Token Count	925993	925993	925993	925993	925993	925993	925993
Y	Overall Accuracy	0.342169	0.399225	0.429509	0.574258	0.683883	0.813410	0.900837
	False Positive Rate	0.000000	0.000000	0.000000	0.000059	0.000063	0.000000	0.000000
	False Negative Rate	1.000000	0.913038	0.866976	0.645716	0.478168	0.281570	0.149947
	Ham Messages	17750	17656	17569	16822	15888	13282	9102
	False Positives	0	0	0	1	1	0	0
	Spam Messages	34125	33969	33806	32553	30987	26093	17773
	False Negatives	34125	31015	29309	21020	14817	7347	2665
	Avg DB Token Count	1449288	1449288	1449288	1449288	1449288	1449288	1449288
SA	Overall Accuracy	0.686972	0.692238	0.699680	0.826710	0.884428	0.944705	0.970989
	False Positive Rate	0.000000	0.000049	0.000000	0.000105	0.000348	0.001078	0.000848
	False Negative Rate	1.000000	0.982624	0.957805	0.550660	0.365614	0.171246	0.088676
	Ham Messages	20750	20574	20391	18990	17254	12064	3536
	False Positives	0	1	0	2	6	13	3
	Spam Messages	9455	9381	9314	8715	7951	5641	1669
	False Negatives	9455	9218	8921	4799	2907	966	148
	Avg DB Token Count	915982	915982	915982	915982	915982	915982	915982

Table 4.5: TEFT (non-corrective) Tests

Finally, Table 4.5 shows what a system of no user interaction offers. I expected accuracy to be dreadful, and it sometimes was. For example, corpus *Y* only reached a 90% overall accuracy rate with a full 5000 message training set. Surprisingly, even though its overall accuracy was much lower at each training set size, *Y*'s false positives were many fewer with non-corrective *TEFT* than either *TEFT-Corrective* or *TOE* for most training set sizes. Corpus *SA* followed *Y*'s trend of fewer false positives with non-corrective *TEFT* compared to *TEFT-Corrective* and *TOE*. With the 5000 message training set, *SA* almost matched its accuracy with *TEFT-Corrective* and *TOE*. Compared to the other training modes, corpus *X* maintained decent overall accuracy, but with an unacceptable level of increased false positives. With no initial training, non-corrective *TEFT* classifies every message as ham. When the first message arrives to be classified, and the filter has no prior knowledge, the filter must assume the message is ham (to avoid false positives). Subsequently, since the filter thinks it has only seen ham before, the next message will also be judged as ham. This will continue for all messages since no errors are corrected.

No matter the training mode, more initial training data generally resulted in fewer false positives and a higher overall accuracy. Trivially, if a user has saved messages, they all should be used to create the initial database. In the event the user has not saved many messages, adequate accuracy

can still be had. *TEFT-Corrective* is recommended when dealing with small initial training sets. If disk space is a major concern, *TOE* gives acceptable accuracy while keeping database size low. The non-corrective method, *TEFT*, is intriguing. False positives were very low with two corpora, and overall accuracy was decent with a 5000 message initial training set. Users don't always catch all mistakes, so the performance of *TEFT* is encouraging. However, non-corrective *TEFT* is not recommended unless user feedback is impossible for a system.

4.3 Weighted Token Probability Function

4.3.1 Establishing the Desire for Weighting

Tests given in Tables 4.6 and 4.7 are designed to show why I believed weighting header data and phrase tokens differently might be beneficial. These tests use the *Base* configuration in Table 4.1, with the changes described. With the *All*, *Normal*, and *No-X* options, only those header lines were tokenized and marked. When a particular subset of headers is used, only those header lines are tokenized. For example, in the *No-X* tests below, all *X*- headers are ignored on input. Two variations of *None* were also run. *None Marked* considers when all headers are tokenized, but the tokens are not marked as having come from headers. Their counts are combined with body tokens. *None Tokenized* only uses body tokens, and the headers are discarded.

Corpus		All	Normal	No-X	None Marked	None Tokenized
X	Overall Accuracy	0.997604	0.996476	0.997745	0.997674	0.975123
	False Positive Rate	0.004796	0.005709	0.004567	0.005937	0.013930
	False Negative Rate	0.001325	0.002548	0.001223	0.000713	0.029763
	Ham Messages	4379	4379	4379	4379	4379
	False Positives	21	25	20	26	61
	Spam Messages	9811	9811	9811	9811	9811
	False Negatives	13	25	12	7	292
Y	Overall Accuracy	0.975926	0.957135	0.972465	0.938047	0.845544
	False Positive Rate	0.001428	0.000439	0.000659	0.000220	0.000330
	False Negative Rate	0.035672	0.064592	0.041299	0.093569	0.233388
	Ham Messages	9102	9102	9102	9102	9102
	False Positives	13	4	6	2	3
	Spam Messages	17773	17773	17773	17773	17773
	False Negatives	634	1148	734	1663	4148
SA	Overall Accuracy	0.980596	0.980596	0.980211	0.980788	0.964073
	False Positive Rate	0.000848	0.001131	0.000848	0.000283	0.001980
	False Negative Rate	0.058718	0.058119	0.059916	0.059317	0.107849
	Ham Messages	3536	3536	3536	3536	3536
	False Positives	3	4	3	1	7
	Spam Messages	1669	1669	1669	1669	1669
	False Negatives	98	97	100	99	180

Table 4.6: Maximum Phrase Length of 1

Corpus		All	Normal	No-X	None Marked	None Tokenized
X	Overall Accuracy	0.997674	0.997463	0.997604	0.998097	0.983369
	False Positive Rate	0.003197	0.003197	0.002969	0.003197	0.013702
	False Negative Rate	0.001937	0.002242	0.002140	0.001325	0.017939
	Ham Messages	4379	4379	4379	4379	4379
	False Positives	14	14	13	14	60
	Spam Messages	9811	9811	9811	9811	9811
	False Negatives	19	22	21	13	176
Y	Overall Accuracy	0.957730	0.947349	0.956428	0.941060	0.880819
	False Positive Rate	0.000000	0.000000	0.000000	0.000000	0.000000
	False Negative Rate	0.063917	0.079615	0.065886	0.089124	0.180217
	Ham Messages	9102	9102	9102	9102	9102
	False Positives	0	0	0	0	0
	Spam Messages	17773	17773	17773	17773	17773
	False Negatives	1136	1415	1171	1584	3203
SA	Overall Accuracy	0.974063	0.974063	0.974448	0.973295	0.967531
	False Positive Rate	0.000848	0.000848	0.000848	0.000848	0.000566
	False Negative Rate	0.079089	0.079089	0.077891	0.081486	0.100060
	Ham Messages	3536	3536	3536	3536	3536
	False Positives	3	3	3	3	2
	Spam Messages	1669	1669	1669	1669	1669
	False Negatives	132	132	130	136	167

Table 4.7: Maximum Phrase Length of 2

First, comparing Table 4.6 to Table 4.7, the difference between a maximum phrase length of two and one is very pronounced. Using pairs of tokens resulted in a substantial decrease of false positives with two corpora. For example, corpus *Y* never experienced a single false positive with pairs (Table 4.7), but had up to thirteen with single-word tokens (Table 4.6). When marking sets of headers and using pairs of words, the overall accuracy of corpus *Y* was slightly lower, but the false positive rate of zero more than made up for it. For corpora *X* and *Y*, increasing the maximum phrase length from one to two gave a reduction, often substantial, of false positives in every test configuration, while maintaining a strong (low) false negative rate. From these results, it appears the suggestion of weighting phrase tokens higher than single-word tokens is justified.

Testing the three different sets of marked header lines (*ALL*, *Normal*, and *No-X*) gave inconclusive results with a maximum phrase length of one. In the *X* and *SA* corpora, *ALL* and *No-X* gave almost identical results. The *Normal* set of headers gave more false positives in *X* than did *ALL* or *No-X*. Corpus *Y* experienced its highest overall accuracy with *ALL*, but *Normal* gave fewer false positives than did *ALL*. The situation is slightly different with a maximum phrase length of two. The three marked header sets gave practically no differences with the *X* and *SA* corpora. Corpus *Y* had significantly more false negatives with *Normal* than with *ALL* or *No-X*, which does not seem surprising, because *Normal* utilizes the fewest header fields, and I assume more data equals better accuracy.

I expected the difference between marking header and not marking to be substantial. I expected marking headers (no matter the set) to give a noticeable increase in overall accuracy and help reduce false positives. As it turned out, the results of *None Marked* were usually comparable to the different marked sets or slightly better in some cases.

For relatively small numbers of counts, the standard deviation in a number of counts is approximately equal to the square root of the number of counts: $\sigma \approx \sqrt{n}$. Statistical significance requires a difference of two, or preferably three or more, standard deviations. In Table 4.6, corpus X had 21 false positives when marking ALL header fields. The standard deviation on 21 is less than 5, therefore the difference between 21 false positives with ALL and 25, 20, and 26 with *Normal*, *No-X*, and *None Marked*, respectively, is not significant. However, the difference between 21 false positives with ALL and 61 with *None Tokenized* is significant. The low accuracy of *None Tokenized* was predicted. Relative to any tested scheme of header marking, ignoring the headers completely gave significantly worse accuracy and especially hurt the false positive rate.

4.3.2 Exploring the *eps* Value

Before header or phrase weights can be tested, we must see how different *eps* values affect accuracy. The configuration for these tests differs from the *Base* configuration in Table 4.1 only by the use of the weighted token probability function introduced in Section 3.3. This function has a constant *eps* that removes the need for hard limits on token probabilities. It was unknown how different *eps* values would affect accuracy. Results of several *eps* values are shown in Table 4.8. The header and phrase weights were set to 1.0. As shown, each corpus behaved differently. Corpus Y is the easiest to read, as it never experienced a single false positive. Its false negatives dropped solidly as *eps* decreased. Corpus X mostly followed the same trend of decreasing false negatives as *eps* decreased, but false positives showed a slight increase, then dropped at the lowest *eps* value. Finally, corpus SA actually saw increased false negatives as *eps* decreased, but a sharp decrease in false positives as *eps* decreased produced desirable results. Due to the differing behaviors, subsequent weighting tests were run with *eps* values of 0.5 and 0.000001. Tested separately, header and phrase weights of 0.5, 0.9, 1.0, 1.5, 2.0, 5.0, 10.0, and 100.0 were tried.

On a side note, benefits of the weighted token probability function are already visible. As shown in Table 4.9, compared to the *Base* configuration, the weighted token probability function with *eps* of 0.000001 increased the overall accuracy of all three corpora. Also, X's false positives were fewer.

Corpus		*eps* Value 1.0	0.5	0.1	0.01	0.0001	0.000001
X	Overall Accuracy	0.989570	0.994644	0.997040	0.997886	0.997745	0.998097
	False Positive Rate	0.002512	0.002512	0.002969	0.003197	0.002969	0.002055
	False Negative Rate	0.013964	0.006625	0.002956	0.001631	0.001937	0.001835
	Ham Messages	4379	4379	4379	4379	4379	4379
	False Positives	11	11	13	14	13	9
	Spam Messages	9811	9811	9811	9811	9811	9811
	False Negatives	137	65	29	16	19	18
Y	Overall Accuracy	0.788540	0.852205	0.931870	0.958140	0.962530	0.964353
	False Positive Rate	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
	False Negative Rate	0.319755	0.223485	0.103021	0.063298	0.056659	0.053902
	Ham Messages	9102	9102	9102	9102	9102	9102
	False Positives	0	0	0	0	0	0
	Spam Messages	17773	17773	17773	17773	17773	17773
	False Negatives	5683	3972	1831	1125	1007	958
SA	Overall Accuracy	0.992123	0.993660	0.993660	0.992315	0.991739	0.985783
	False Positive Rate	0.006787	0.004808	0.002545	0.002545	0.001697	0.001131
	False Negative Rate	0.010186	0.009587	0.014380	0.018574	0.022169	0.041941
	Ham Messages	3536	3536	3536	3536	3536	3536
	False Positives	24	17	9	9	6	4
	Spam Messages	1669	1669	1669	1669	1669	1669
	False Negatives	17	16	24	31	37	70

Table 4.8: *eps* Tests

Corpus		Base	Weighted Token Probability Function, eps of 0.000001
X	Overall Accuracy	0.997674	0.998097
	False Positive Rate	0.003197	0.002055
	False Negative Rate	0.001937	0.001835
	Ham Messages	4379	4379
	False Positives	14	9
	Spam Messages	9811	9811
	False Negatives	19	18
Y	Overall Accuracy	0.957730	0.964353
	False Positive Rate	0.000000	0.000000
	False Negative Rate	0.063917	0.053902
	Ham Messages	9102	9102
	False Positives	0	0
	Spam Messages	17773	17773
	False Negatives	1136	958
SA	Overall Accuracy	0.974063	0.985783
	False Positive Rate	0.000848	0.001131
	False Negative Rate	0.079089	0.041941
	Ham Messages	3536	3536
	False Positives	3	4
	Spam Messages	1669	1669
	False Negatives	132	70

Table 4.9: Base and *eps* of 0.000001 Results

4.3.3 Header Weights

In these tests, all header lines were tokenized and marked. During classification, the weighted token probability function applied the given weight to all header tokens. Weights > 1.0 strengthen the token's probability, and weights < 1.0 weaken the token's probability. For example, if a token has a probability of 0.9, a weight > 1.0 will strengthen that probability, pushing it closer to 1.0. The remaining tokens (all body tokens) are given the weight of 1.0. The phrase weight was also left unchanged at 1.0. As was mentioned earlier, my belief was that header data is more important than body data. I expected higher header weights to result in increased accuracy (while maintaining a low rate of false positives).

Results for header weights ≤ 1.0 are shown in Table 4.10. Compared to an *eps* value of 0.5, 0.000001 gave much less movement as the header weight was changed. In other words, with *eps* of 0.000001, all header weights resulted in very similar results. For example, corpus *Y*, as expected, went from 4679 to 3972 false negatives as the weight increased using *eps* of 0.5. However, with *eps* of 0.000001, false negatives in corpus *Y* decreased very slightly from 1000 to 958. The other corpora showed similar stagnant results with *eps* of 0.000001.

Table 4.11 shows the effects of further raising the header weight beyond 1.0. Again, corpus *Y* proved to be very cooperative as its false negatives sharply decreased with increasing header weights with *eps* of 0.5. With *eps* of 0.000001, its false negatives also decreased with increasing header weights, but at a slower rate. Also, two false positives made an appearance in the *Y* corpus with the highest tested header weight and an *eps* value of 0.000001. No matter the *eps* value, corpus *X* showed a trend of slightly increasing false positives as the header weight increased. Finally, as the header weight increased, corpus *SA* showed acceptable increases of false negatives due to decreasing false positives with *eps* of 0.5. Under *eps* of 0.000001, the same corpus showed no change, no matter the header weight.

Overall, the tested separate header weight configurations cannot be fully recommended. Increased weights showed increased overall accuracy in most cases, but some corpora also showed a trend of increasing false positives. The possibility of increased false positives is not a chance to be taken lightly. However, if maximum overall accuracy is desired without regard to false positives, *eps* of 0.000001 with the highest tested header weight did perform slightly better than the *Base* configuration in all three corpora.

Corpus		Header Weight 0.5	0.9	1.0
'eps': 0.5				
X	Overall Accuracy	0.993728	0.994644	0.994644
	False Positive Rate	0.002284	0.002284	0.002512
	False Negative Rate	0.008052	0.006727	0.006625
	Ham Messages	4379	4379	4379
	False Positives	10	10	11
	Spam Messages	9811	9811	9811
	False Negatives	79	66	65
Y	Overall Accuracy	0.825898	0.847665	0.852205
	False Positive Rate	0.000000	0.000000	0.000000
	False Negative Rate	0.263265	0.230349	0.223485
	Ham Messages	9102	9102	9102
	False Positives	0	0	0
	Spam Messages	17773	17773	17773
	False Negatives	4679	4094	3972
SA	Overall Accuracy	0.992891	0.993660	0.993660
	False Positive Rate	0.005939	0.005090	0.004808
	False Negative Rate	0.009587	0.008987	0.009587
	Ham Messages	3536	3536	3536
	False Positives	21	18	17
	Spam Messages	1669	1669	1669
	False Negatives	16	15	16
'eps': 0.000001				
X	Overall Accuracy	0.998097	0.998027	0.998097
	False Positive Rate	0.002055	0.002055	0.002055
	False Negative Rate	0.001835	0.001937	0.001835
	Ham Messages	4379	4379	4379
	False Positives	9	9	9
	Spam Messages	9811	9811	9811
	False Negatives	18	19	18
Y	Overall Accuracy	0.962791	0.964130	0.964353
	False Positive Rate	0.000000	0.000000	0.000000
	False Negative Rate	0.056265	0.054240	0.053902
	Ham Messages	9102	9102	9102
	False Positives	0	0	0
	Spam Messages	17773	17773	17773
	False Negatives	1000	964	958
SA	Overall Accuracy	0.985783	0.985591	0.985783
	False Positive Rate	0.000848	0.001131	0.001131
	False Negative Rate	0.042540	0.042540	0.041941
	Ham Messages	3536	3536	3536
	False Positives	3	4	4
	Spam Messages	1669	1669	1669
	False Negatives	71	71	70

Table 4.10: Header Weights ≤ 1.0

Corpus		Header Weight 1.5	2.0	5.0	10.0	100.0
		'eps': 0.5				
X	Overall Accuracy	0.994715	0.995349	0.996265	0.996476	0.996899
	False Positive Rate	0.002740	0.002740	0.002969	0.003425	0.003425
	False Negative Rate	0.006421	0.005504	0.004077	0.003567	0.002956
	Ham Messages	4379	4379	4379	4379	4379
	False Positives	12	12	13	15	15
	Spam Messages	9811	9811	9811	9811	9811
	False Negatives	63	54	40	35	29
Y	Overall Accuracy	0.871033	0.881898	0.910735	0.926847	0.947163
	False Positive Rate	0.000000	0.000000	0.000000	0.000000	0.000000
	False Negative Rate	0.195015	0.178585	0.134980	0.110617	0.079896
	Ham Messages	9102	9102	9102	9102	9102
	False Positives	0	0	0	0	0
	Spam Messages	17773	17773	17773	17773	17773
	False Negatives	3466	3174	2399	1966	1420
SA	Overall Accuracy	0.994236	0.995005	0.994813	0.995005	0.994236
	False Positive Rate	0.003676	0.003111	0.002828	0.002262	0.001980
	False Negative Rate	0.010186	0.008987	0.010186	0.010785	0.013781
	Ham Messages	3536	3536	3536	3536	3536
	False Positives	13	11	10	8	7
	Spam Messages	1669	1669	1669	1669	1669
	False Negatives	17	15	17	18	23
		'eps': 0.000001				
X	Overall Accuracy	0.997956	0.997956	0.998027	0.998168	0.998379
	False Positive Rate	0.002284	0.002284	0.002512	0.002512	0.002740
	False Negative Rate	0.001937	0.001937	0.001733	0.001529	0.001121
	Ham Messages	4379	4379	4379	4379	4379
	False Positives	10	10	11	11	12
	Spam Messages	9811	9811	9811	9811	9811
	False Negatives	19	19	17	15	11
Y	Overall Accuracy	0.965730	0.966288	0.969005	0.970753	0.975367
	False Positive Rate	0.000000	0.000000	0.000220	0.000220	0.000220
	False Negative Rate	0.051820	0.050976	0.046756	0.044112	0.037135
	Ham Messages	9102	9102	9102	9102	9102
	False Positives	0	0	2	2	2
	Spam Messages	17773	17773	17773	17773	17773
	False Negatives	921	906	831	784	660
SA	Overall Accuracy	0.985975	0.986167	0.985975	0.985975	0.986167
	False Positive Rate	0.001131	0.001131	0.001131	0.001131	0.001131
	False Negative Rate	0.041342	0.040743	0.041342	0.041342	0.040743
	Ham Messages	3536	3536	3536	3536	3536
	False Positives	4	4	4	4	4
	Spam Messages	1669	1669	1669	1669	1669
	False Negatives	69	68	69	69	68

Table 4.11: Header Weights Tests > 1.0

4.3.4 Phrase Weights

Phrase weight tests were conducted with the same configuration as the header weight tests. All single-word tokens were given unit weight of 1.0. All other tokens (which are of phrase size > 1), regardless of whether or not they are in a header field, are given the specified weight. Note the maximum phrase length of these tests was set to two, so only pairs of words were weighted. Weights applied to larger phrases were not tested. The header weight was left constant at 1.0. As with the header weights, phrase weights gave mixed results.

Phrase weights ≤ 1.0 are shown in Table 4.12. Neither *eps* value gave the same sort of movement from changing phrase weights that changing header weights gave. Only corpus Y gave conclusive results as its false negatives decreased by almost 500 with *eps* of 0.5.

In Table 4.13, corpus X once again showed a trend of increasing false positives as the phrase weight increased. Also, again corpus SA showed decreasing false positives with *eps* of 0.5 at the expensive of higher false negatives, as the phrase weights increased. Corpus Y continued its downward trend of false negatives under *eps* of 0.5 as phrase weights increased, but showed little change under *eps* of 0.000001.

As with the header weight results, an *eps* value of 0.000001 resulted in either higher overall accuracy or decreased false positives compared to *eps* of 0.5. Again, with *eps* of 0.000001, results changed very little with changes in the phrase weight. Overall, the conclusions are much the same as with header weights. Since corpus X experienced increasing false positives as the phrase weights increased, increasing phrase weights cannot be recommended.

4.4 Miscellaneous Tests

Other interesting test results are shown in Table 4.14. All tests are based on the *Base* configuration.

Robinson's Geometric Mean combined probability function was tested. This configuration differed from *Base* only by the use of that function instead of *Base's* SP-Graham combined probability function. The *Geometric Mean* setup did not give a single false positive, but overall accuracy was substantially lower. This test, conducted at the usual spam threshold of 0.7, showed a terrible false negative rate for each corpus. Therefore, another test was run at a lower threshold of 0.6. This decision threshold showed a much improved false negative rate, but still far from the accuracy of *Base*. Out of fairness, the *Base* configuration was also tested with a threshold of 0.6. This test showed an additional false positive in both X and Y, so 0.7 is favored for the *Base* setup.

Corpus		Phrase Weight 0.5	0.9	1.0
'eps': 0.5				
X	Overall Accuracy	0.993587	0.994433	0.994644
	False Positive Rate	0.002740	0.002740	0.002512
	False Negative Rate	0.008052	0.006829	0.006625
	Ham Messages	4379	4379	4379
	False Positives	12	12	11
	Spam Messages	9811	9811	9811
	False Negatives	79	67	65
Y	Overall Accuracy	0.834753	0.848558	0.852205
	False Positive Rate	0.000000	0.000000	0.000000
	False Negative Rate	0.249873	0.228999	0.223485
	Ham Messages	9102	9102	9102
	False Positives	0	0	0
	Spam Messages	17773	17773	17773
	False Negatives	4441	4070	3972
SA	Overall Accuracy	0.994044	0.993852	0.993660
	False Positive Rate	0.003676	0.004525	0.004808
	False Negative Rate	0.010785	0.009587	0.009587
	Ham Messages	3536	3536	3536
	False Positives	13	16	17
	Spam Messages	1669	1669	1669
	False Negatives	18	16	16
'eps': 0.000001				
X	Overall Accuracy	0.997956	0.998097	0.998097
	False Positive Rate	0.002055	0.002284	0.002055
	False Negative Rate	0.002039	0.001733	0.001835
	Ham Messages	4379	4379	4379
	False Positives	9	10	9
	Spam Messages	9811	9811	9811
	False Negatives	20	17	18
Y	Overall Accuracy	0.965247	0.964242	0.964353
	False Positive Rate	0.000000	0.000000	0.000000
	False Negative Rate	0.052552	0.054071	0.053902
	Ham Messages	9102	9102	9102
	False Positives	0	0	0
	Spam Messages	17773	17773	17773
	False Negatives	934	961	958
SA	Overall Accuracy	0.985783	0.985783	0.985783
	False Positive Rate	0.000848	0.001131	0.001131
	False Negative Rate	0.042540	0.041941	0.041941
	Ham Messages	3536	3536	3536
	False Positives	3	4	4
	Spam Messages	1669	1669	1669
	False Negatives	71	70	70

Table 4.12: Phrase Weights ≤ 1.0

Corpus		Phrase Weight 1.5	2.0	5.0	10.0	100.0
		'eps': 0.5				
X	Overall Accuracy	0.995208	0.995631	0.996406	0.996476	0.997393
	False Positive Rate	0.002284	0.002512	0.002740	0.003197	0.003197
	False Negative Rate	0.005912	0.005198	0.003975	0.003669	0.002344
	Ham Messages	4379	4379	4379	4379	4379
	False Positives	10	11	12	14	14
	Spam Messages	9811	9811	9811	9811	9811
	False Negatives	58	51	39	36	23
Y	Overall Accuracy	0.865414	0.874530	0.896856	0.906419	0.922047
	False Positive Rate	0.000000	0.000000	0.000000	0.000000	0.000000
	False Negative Rate	0.203511	0.189726	0.155967	0.141507	0.117875
	Ham Messages	9102	9102	9102	9102	9102
	False Positives	0	0	0	0	0
	Spam Messages	17773	17773	17773	17773	17773
	False Negatives	3617	3372	2772	2515	2095
SA	Overall Accuracy	0.993660	0.993084	0.993468	0.993276	0.992699
	False Positive Rate	0.004808	0.005373	0.003959	0.003676	0.003394
	False Negative Rate	0.009587	0.010186	0.011983	0.013182	0.015578
	Ham Messages	3536	3536	3536	3536	3536
	False Positives	17	19	14	13	12
	Spam Messages	1669	1669	1669	1669	1669
	False Negatives	16	17	20	22	26
		'EPS': 0.000001				
X	Overall Accuracy	0.997956	0.998027	0.998027	0.998027	0.997956
	False Positive Rate	0.002055	0.002284	0.002284	0.002284	0.002512
	False Negative Rate	0.002039	0.001835	0.001835	0.001835	0.001835
	Ham Messages	4379	4379	4379	4379	4379
	False Positives	9	10	10	10	11
	Spam Messages	9811	9811	9811	9811	9811
	False Negatives	20	18	18	18	18
Y	Overall Accuracy	0.963088	0.963163	0.963163	0.963163	0.962456
	False Positive Rate	0.000000	0.000000	0.000000	0.000000	0.000000
	False Negative Rate	0.055815	0.055702	0.055702	0.055702	0.056772
	Ham Messages	9102	9102	9102	9102	9102
	False Positives	0	0	0	0	0
	Spam Messages	17773	17773	17773	17773	17773
	False Negatives	992	990	990	990	1009
SA	Overall Accuracy	0.985207	0.985399	0.984822	0.984822	0.984630
	False Positive Rate	0.001131	0.001414	0.001131	0.001131	0.001131
	False Negative Rate	0.043739	0.042540	0.044937	0.044937	0.045536
	Ham Messages	3536	3536	3536	3536	3536
	False Positives	4	5	4	4	4
	Spam Messages	1669	1669	1669	1669	1669
	False Negatives	73	71	75	75	76

Table 4.13: Phrase Weights > 1.0

Corpus		Base	Base 0.6	Geometric Mean	Geometric Mean 0.6	Triples	Whole Message Matrix
X	Overall Accuracy	0.997674	0.997815	0.946723	0.978858	0.996688	0.986258
	False Positive Rate	0.003197	0.003425	0.000000	0.000000	0.002512	0.000457
	False Negative Rate	0.001937	0.001631	0.077056	0.030578	0.003669	0.019672
	Ham Messages	4379	4379	4379	4379	4379	4379
	False Positives	14	15	0	0	11	2
	Spam Messages	9811	9811	9811	9811	9811	9811
	False Negatives	19	16	756	300	36	193
	Avg DB Token Count	925993	925993	925993	925993	1786768	925993
Y	Overall Accuracy	0.957730	0.962158	0.781805	0.835684	0.939498	0.784037
	False Positive Rate	0.000000	0.000110	0.000000	0.000000	0.000549	0.000000
	False Negative Rate	0.063917	0.057165	0.329939	0.248467	0.091206	0.326563
	Ham Messages	9102	9102	9102	9102	9102	9102
	False Positives	0	1	0	0	5	0
	Spam Messages	17773	17773	17773	17773	17773	17773
	False Negatives	1136	1016	5864	4416	1621	5804
	Avg DB Token Count	1449288	1449288	1449288	1449288	3205845	1449288
SA	Overall Accuracy	0.974063	0.975985	0.889145	0.918540	0.970221	0.941402
	False Positive Rate	0.000848	0.000848	0.000000	0.000000	0.001414	0.000848
	False Negative Rate	0.079089	0.073098	0.345716	0.254044	0.089874	0.180947
	Ham Messages	3536	3536	3536	3536	3536	3536
	False Positives	3	3	0	0	5	3
	Spam Messages	1669	1669	1669	1669	1669	1669
	False Negatives	132	122	577	424	150	302
	Avg DB Token Count	915982	915982	915982	915982	1932047	915982

Table 4.14: Miscellaneous Tests

The *Triples* test used a maximum phrase length of three. The lower accuracy of was unexpected. Just as pairs performed better than single word tokens, I assumed the more data gathered by triples would equal higher accuracy. Actually, it appears triples were more susceptible to *word salad* – the insertion of unrelated, seemingly hammy words in an attempt to dilute a message's spamminess. Table 4.15 shows the decision matrix used from a word salad spam message. Using triples created more tokens from the word salad, and they succeeded in appearing hammy. Since the decision matrix building process always favors hammy tokens, the hammy triples forced other spammy tokens out. With triples this spam was classified as ham, but correctly classified using pairs. Obviously, triples cannot be recommended if disk space is a concern. If triples were to be used, a larger decision matrix might help. The matrix size of 27 is approximately optimal for a maximum phrase size of two tokens, according to Brian Burton, but may be too large for a phrase size of one and too small for a size of three.

The *Whole Message Matrix* test differed from *Base* by using a decision matrix of size 1,000,000 and a max token usage count of 1,000,000. This should have effectively included all of a message's tokens in the decision matrix. Pairs of words were still used. Corpus Y saw a serious increase of false negatives. This could be due to successful word salad attacks. The decrease of false positives in X with *Whole Message Matrix* relative to *Base* is a welcome change.

Triples				Pairs			
Ham Count	Spam Count	Score	Token	Ham Count	Spam Count	Score	Token
5	0	0.000001	paled	7	0	0.000001	face and
7	0	0.000001	face and	10	0	0.000001	irradiated
8	0	0.000001	then i ll	5	0	0.000001	paled
5	0	0.000001	first or	5	0	0.000001	first or
5	0	0.000001	ll take	5	0	0.000001	ll take
7	0	0.000001	show him	17	0	0.000001	secretary of
10	0	0.000001	irradiated	6	0	0.000001	my neck
13	0	0.000001	secretary of state	6	0	0.000001	annals of
17	0	0.000001	secretary of	7	0	0.000001	show him
6	0	0.000001	out by the	6	0	0.000001	myself with
6	0	0.000001	my neck	7	0	0.000001	brass
25	0	0.000001	think i am	5	0	0.000001	really the
6	0	0.000001	annals of	0	15	0.999999	lordship
6	0	0.000001	myself with	0	15	0.999999	lordship
7	0	0.000001	brass	0	7	0.999999	stared
6	0	0.000001	don t say	0	14	0.999999	rebels
9	0	0.000001	more than the	0	10	0.999999	just try
5	0	0.000001	really the	0	5	0.999999	itself a
0	15	0.999999	lordship	0	18	0.999999	try us
0	15	0.999999	lordship	0	10	0.999999	levasseur
0	5	0.999999	Hsubject_she	0	7	0.999999	thee
0	7	0.999999	mr blood	0	7	0.999999	mr blood
0	7	0.999999	thee	0	10	0.999999	ll show
0	11	0.999999	get top	0	9	0.999999	king s
0	7	0.999999	you get top	0	13	0.999999	his lordship
0	6	0.999999	Hsubject_i m	0	5	0.999999	Hsubject_she
0	9	0.999999	king s	0	14	0.999999	land and

Table 4.15: Triples vs Pairs Matrices

The *Geometric Mean 0.6* and *Whole Message Matrix* results test the definition of how results are compared. Both setups gave equal or better false positive rates than *Base*, but their false negative rate (and overall accuracy) is at times significantly worse. For example, in corpus *X*, *Geometric Mean 0.6* gave zero false positives compared to fourteen for *Base*, but it gave a false negative rate of 3.06% compared to 0.19% with Base. The situation is much clearer with corpus *Y*, as neither setup gave false positives, but *Base* gave an obviously better false negative rate. Which configuration is 'better'? The answer depends on the user. If a particular setup gives the highest overall accuracy, it is not necessarily better than another. A low rate of false positives is extremely important.

Chapter 5

Summary, Conclusions and Future Work

5.1 Summary

Statistical spam filtering, inspired from Paul Graham's original essay [6], is a relatively new and successful technique to free users' inboxes from spam. The procedure is straight-forward:

- An initial database is built.
  - Saved ham and spam are broken into tokens.
  - A token database is built, with ham and spam counts for each token.
- New messages are classified.
  - The message is tokenized.
  - An individual probability for each token is calculated.
  - The combined probability for the message, whether or not it is spam, is calculated.
  - The tokens from the message might be added to the database.
  - Error correction may be done later by the user.

This system of filtering requires only the pre-classified sets of ham and spam. Automatic learning through statistical analysis of the token database gives a low rate of errors.

There are a few major modules in a statistical spam filter. The tokenizer is responsible for breaking messages into tokens. This determines the actual information that the filter will see. The

database will be large and must give fast and accurate access. The entire message is usually not used for classification. Instead, a smaller decision matrix of tokens is built. The decision matrix is fed to the combined probability function and a decision is made. Finally, after classification, different methods of training (updating the database) may be employed.

In an effort to study the benefits of different techniques, a general test system was designed and implemented in this paper. This System gives many options:

- Tokenization: The tokenizer uses code from the open-source SpamProbe project.
  - Marking header tokens is a common technique that is implemented. Tokens are prefixed with the name of the header field they are found in.
  - Word phrases are implemented. Instead of just single-word tokens, n-word tokens are gathered from messages.
- Token Probability Function:
  - Paul Graham's original in Figure 2.2.
  - A new weighted individual token probability function was created (see Section 3.3). With this function, weights can be applied to header and phrase tokens to give them stronger or weaker scores. Also, hard limits on token probabilities are eliminated.
- Decision Matrix:
  - Variable window size.
  - Variable number of token repeats allowed.
- Combination Functions:
  - Graham's original in Figure 2.4.
  - SpamProbe's in Figure 2.7.
  - Gary Robinson's geometric mean in Figure 2.8.
- Post-Classification Training:
  - Corrective TEFT: Every message is added to the database, and corrections are immediately applied.
  - Non-Corrective TEFT: Every message is added to the database. No corrections are made.
  - TOE: Only misclassified messages are trained. Errors are immediately corrected.

Many filter configurations were tested. The *Base* configuration in Table 4.1 is similar to the defaults given by the popular spam filter SpamProbe. This was tested against a setup similar to Graham's original model. The *Base* setup used a two-word maximum phrase length and marked header tokens, whereas the *Graham-like* model used single-word phrases and did not mark header tokens. The results of these tests are in Table 5.1. Both models performed well. Due to the lack of

		Configuration			
Corpus		Base	Graham-like	Singles	Triples
X	Overall Accuracy	0.997674	0.998450	0.997604	0.996688
	False Positive Rate	0.003197	0.003197	0.004796	0.002512
	False Negative Rate	0.001937	0.000815	0.001325	0.003669
	Avg DB Token Count	925993	225921	322000	1786768
Y	Overall Accuracy	0.957730	0.943107	0.975926	0.939498
	False Positive Rate	0.000000	0.000220	0.001428	0.000549
	False Negative Rate	0.063917	0.085917	0.035672	0.091206
	Avg DB Token Count	1449288	295528	419885	3205845
SA	Overall Accuracy	0.974063	0.967723	0.980596	0.970221
	False Positive Rate	0.000848	0.000000	0.000848	0.001414
	False Negative Rate	0.079089	0.100659	0.058718	0.089874
	Avg DB Token Count	915982	189235	271452	1932047

Table 5.1: Base, Graham-like, Singles, and Triples Summary

two-word phrases and not marking header tokens, the *Graham-like* system produces databases with substantially fewer tokens.

Table 5.1 also compares three different maximum phrase lengths. The *Singles* and *Triples* setups differ from *Base* only by their maximum phrase lengths. The *Base* setup always used a maximum phrase length of two, and it proved to be most effective, at least when using a decision matrix size of 27. The overall accuracy of *Singles* was higher than *Base* with two corpora, but *Singles* also gave a higher false positive rate with two corpora. Compared to the two-word phrase model of *Base*, *Triples* gave lower overall accuracy in all three corpora, and a higher false positive rate with two corpora.

Disk space is a common concern for many users. A token database can easily exceed 1,000,000 tokens. Using a one-word phrase length decreases database size relative to two-word phrases. Database size can also be reduced by only updating the database when errors are corrected. This method is referred to as TOE. The standard method used in the *Base* setup is TEFT-Corrective, where every message is added to the database after classification. In this simulation, using TOE or TEFT-Corrective errors were immediately corrected before the next message classification began. To study the effects of small training set sizes, tests were conducted with zero to 5000 messages in the initial training set.

Figure 5.1 shows the overall accuracy given by TOE and TEFT-Corrective for the three corpora. Looking at this graph alone, the conclusion would be that TOE is better. Corpora *Y* and *SA* both

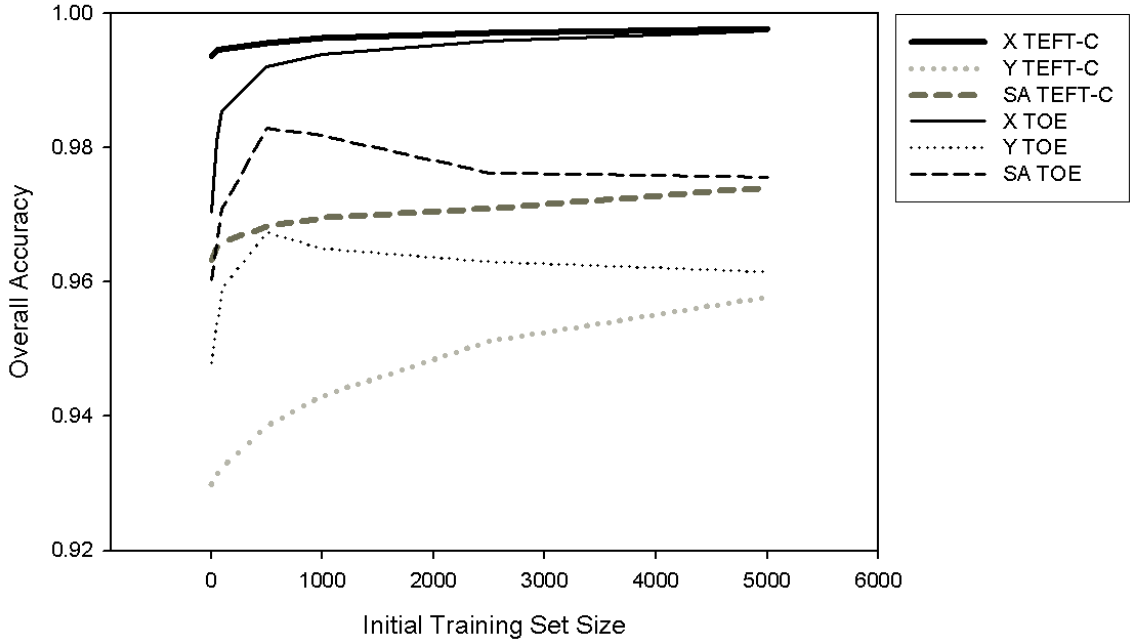

Figure 5.1: TEFT-Corrective and TOE Overall Accuracy

achieve higher overall accuracy with TOE at almost all tested training set sizes. However, Figure 5.2 clearly shows TEFT-Corrective performing better than TOE with regard to false positive rates. For this reason, TEFT-Corrective is the preferred technique. If database size if a major concern, TOE does substantially reduce the token counts, but the higher false positive rate is a concern. Figure 5.1 also shows that great overall accuracy is still given with a small initial training set. Even with zero initial training messages, all three corpora had overall accuracy greater than 92%. TEFT-Corrective performed better than TOE with small training sets.

The new weighted individual token probability function introduced in Section 3.3 was tested. First, different values for *eps* had to be tried. This variable removes the necessity for hard limits on the token probabilities. As shown in Table 4.8, the lowest tested value of *eps*, 0.000001, was favored with the *X* and *Y* corpora. Corpus *SA* saw its highest overall accuracy with *eps* at 0.5 and 0.1. All header and phrase weighting tests were conducted with *eps* of both 0.5 and 0.000001.

As shown in Tables 4.10 and 4.11, mixed results were obtained from weighting header tokens. In these tests, all header lines were tokenized. Any token from a header line was marked as such and

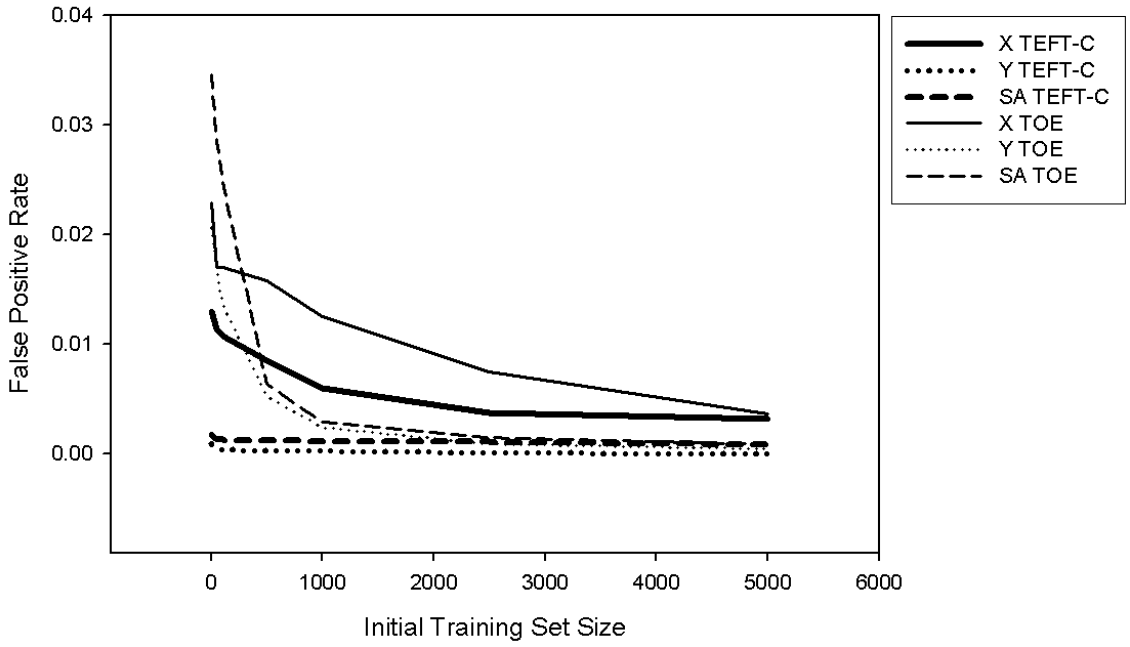

Figure 5.2: TEFT-Corrective and TOE False Positive Rates

weighted. The remaining tokens (those in the body) were given unit weight of 1.0. The weighting technique strengthens a token's probability when the weight is > 1.0, and weakens the probability for weights < 1.0. Corpus Y mostly performed well. As the header weight increased, Y's false negatives steadily decreased. However, two false positives did appear in Y with *eps* of 0.000001 at the two highest tested header weights. No matter the *eps* value, X showed a trend of slightly increasing false positives as the header weight increased. The increased header weights cannot be fully recommended due the action of the X corpus and the false positives in Y.

Tables 4.12 and 4.13 give the results of phrase weights. These tests used a maximum phrase length of two, and only the two-word tokens were weighted. The remaining single-word tokens were given unit weight of 1.0. The results are similar to the header weight tests. Again, corpus X showed a trend of increasing false positives as the phrase weight increased. Due to this motion, increasing the phrase weight cannot be recommended.

Even though increasing or decreasing the header or phrase weights separately did not give conclusive results, the weighted token probability function gave favorable results when weights were left at the default 1.0 value. As shown in Table 5.2, corpora Y and SA achieved their highest overall accuracy with the weighted token probability function using *eps* of 0.000001 compared to the *Base*

Corpus		Base	Graham-like	Weighted Token Probability Function, eps of 0.000001
X	Overall Accuracy	0.997674	0.998450	0.998097
	False Positive Rate	0.003197	0.003197	0.002055
	False Negative Rate	0.001937	0.000815	0.001835
Y	Overall Accuracy	0.957730	0.943107	0.964353
	False Positive Rate	0.000000	0.000220	0.000000
	False Negative Rate	0.063917	0.085917	0.053902
SA	Overall Accuracy	0.974063	0.967723	0.985783
	False Positive Rate	0.000848	0.000000	0.001131
	False Negative Rate	0.079089	0.100659	0.041941

Table 5.2: Base, Graham-like, and *eps* of 0.000001 Summary

and *Graham-like* configurations. Also, with the weighted token probability function, corpus X had a lower false positive rate compared to the other two configurations.

5.2 Conclusions

The *Base* configuration in Table 4.1 performed well. This filter configuration is similar to the defaults given by the popular spam filter SpamProbe. Corpora X, Y, and SA saw overall accuracy of 99.8%, 95.8%, and 97.4%, respectively. Even though corpus Y's overall accuracy of 95.8% was the lowest, this corpus had zero false positives which is very much desired. The false positive rates of X and SA were reasonably low at 0.32% and 0.08%.

Even though it is older and simpler, the *Graham-like* configuration gave results very close to the *Base* setup. The *Graham-like* setup did not use methods now considered to be common-place, such as tokenizing pairs of words and marking header data. Paul Graham introduced an effective system four years ago, and it is still standing strong.

The System presented in this paper with its *Base* configuration thrives when given an abundance of data. However, users with few or no saved messages need not worry. With *TEFT-Corrective* especially, great accuracy can still be had with a very small training set. If disk space is a concern, the *TOE* method of training significantly reduces the database's token count while maintaining high accuracy. Users are encouraged never to assume their spam filter is perfect. The spam message folder should be checked periodically for mistakes.

The new weighted token probability function gave inconclusive results when weighting header data or phrased tokens. When one corpus experienced a sharp decrease in false negatives with decreasing the *eps* value, another corpus showed a trend of increasing false positives. The possibility

of increased false positives is not a risk most users probably want to take. However, when applied with the default weights of 1.0, the weighted tokens probability function with *eps* of 0.000001 gave higher overall accuracy compared to the *Base* configuration.

No matter what configuration was used, each tested corpus seemed to reach an accuracy plateau. *X* consistently maintained 99+% overall accuracy, but false positives were a regular problem. *Y* had trouble breaking 95-96%, but false positives were rarely seen. *SA* reliably gave 96-98% accuracy with a minute false positive rate. This accuracy plateau may be tough to overcome with current technology. The 'plateau at 99.9%' referred to by Yerazunis [20] is much more difficult to achieve for a heterogeneous ham corpus such as *Y*, and probably impossible using the mainstream methods we have applied in this paper.

From this study, the following general recommendations are made:

- Use two-word token phrases.
- Use as many saved messages as possible for initial training.
- The spam message folder should be monitored; false positives are not impossible.
- If a very small initial training set must be used, employ a TEFT-Corrective training system and closely monitor your spam message folder and inbox for mistakes.
- If disk space is a major concern, consider TOE or single-word tokens.
- If a large initial training set is available, try different options to find those that work best with your email.

5.3 Suggestions for Future Work

The header and phrase token weighting function presented in this paper produced mixed results. Certain situations did however show promise. The weighted token probability function could be revised or a new model for weighting could possibly show better results. Another idea is to allow separate weights for separate header fields. For example, weight the *To* and *Subject* fields higher than other fields.

Database growth is an interesting topic. Different database cleanup methods could be studied. A popular cleanup technique is to delete tokens whose combined ham and spam counts are below a threshold and been updated for a certain number of days. The modification date would have to be stored along with each token. Another method is to delete tokens whose counts are below a

threshold, and that haven't been modified for some number of subsequent message classifications. Alternatively, instead of deleting tokens whose counts are below a threshold, we could delete tokens whose counts are above a threshold and probability is near 0.5. This would remove neutral tokens that should never appear in any decision matrix and therefore are not necessary. A further method for database cleanup is to remove entire messages of a certain age from the database. When each message is purged, it would be re-tokenized and all token counts decremented. This could be impractical, since users would have to retain all messages. Ham and spam change over time, and this method would allow the database to move and adapt correspondingly.

Tokenization is a never-ending area of research. Token reconstruction is an interesting technique. Consider the following tokens. They all came from spam in the X corpus.

Pharam acy	Sto ck	[V]-[i]-[a]-[g]-[r]-[a]	re'mo}-[v]â)l]	R|O|L|E|X

Humans easily recognize these tokens as *Pharamacy*, *Stock*, *Viagra*, *removal*, and *ROLEX*, but to the filter they may be useless garbage. John Graham-Cumming, author of the spam filter POPFile, refers to this spammer trick as 'L o s t i n s p a c e.' [8]. A tokenizer could reassemble these excessively delimited tokens. However, a well-trained filter might already recognize single characters as spammy. Another interesting proposed change to the tokenizer is a sliding window. A window of size n moves over the messages, and whatever characters are found in that window form a token.

The decision matrix should be analyzed further. Our base model of 27 tokens with 2 repeats may not be the most accurate. The optimum decision matrix size probably should be different for single, pair, and triple-word tokens.

Multi-user environments present many interesting challenges. Disk space is a common concern. The *TOE* method has shown it successfully limits database growth while maintaining high accuracy, and cleanup methods have been suggested. Another possible solution is a fixed-size database. This could be implemented through an automatic cleanup system. The database would purge tokens as necessary to allow additional new tokens while maintaining a maximum size. A single, shared database could also be investigated. The handling of new users is an interesting topic. When a new email account is created, a generic starter database might give better performance than *TEFT-Corrective* gives with no initial data. This generic database could be built from an assortment of interesting tokens collected from other users' databases.

Bibliography

[1] Androutsopoulos, I., Koutsias, J., Chandrinos, K., Paliouras, G., and Spyropoulos, C. An Evaluation of Naive Bayesian Anti-Spam Filtering. In *Proceedings of the 11th European Conference on Machine Learning* (Barcelona, Spain, 2000), pp. 9–17.

[2] Burton, B. Bayesian Spam Filtering Tweaks. In *Proceedings of the Spam Conference* (2003). Available: http://spamprobe.sourceforge.net/paper.html.

[3] Chandler, J. P. Personal Communication, 2006.

[4] Cormack, G. Standardized Spam Filter Evaluation. In *Proceedings of the Spam Conference* (2005). Available: http://plg.uwaterloo.ca/∼gvcormac/spam/spamconference.

[5] Cormack, G., and Lynam, T. Spam Corpus Creation for TREC. In *Proceedings of the Second Conference on Email and Anti-Spam* (2005). Available: http://www.ceas.cc/papers-2005/162.pdf.

[6] Graham, P. A Plan for Spam, 2002. Available: http://www.paulgraham.com/plan.html.

[7] Graham, P. Better Bayesian Filtering. In *Proceedings of the 2003 Spam Conference* (2003). Available: http://www.paulgraham.com/better.html.

[8] Graham-Cumming, J. The Spammers' Compendium. In *Proceedings of the Spam Conference* (2003). Available: http://popfile.sourceforge.net/SpamConference011703.pdf.

[9] Heckerman, D. Tutorial on Learning in Bayesian Networks. Tech. Rep. MSR-TR-95-06, Microsoft, 1995.

[10] Lowd, D., and Meek, C. Good Word Attacks on Statistical Spam Filters. In *Proceedings of the Second Conference on Email and Anti-Spam* (2005). Available: http://www.ceas.cc/papers-2005/125.pdf.

[11] Negnevitsky, M. *Artificial Intelligence: A Guide to Intelligent Systems.* Addison-Wesley, Harlow, England, 2002.

[12] Pantel, P., and Lin, D. SpamCop: A Spam Classification & Organization Program. In *Learning for Text Categorization: Papers from the 1998 Workshop* (Madison, Wisconsin, 1998), AAAI Technical Report WS-98-05.

[13] Robinson, G. Gary Robinson's Rants. Available: http://www.garyrobinson.net.

[14] Robinson, G. A Statistical Approach to the Spam Problem. *Linux J. 2003*, 107 (2003), 3.

[15] Sahami, M., Dumais, S., Heckerman, D., and Horvitz, E. A Bayesian Approach to Filtering Junk E-Mail. In *Learning for Text Categorization: Papers from the 1998 Workshop* (Madison, Wisconsin, 1998), AAAI Technical Report WS-98-05.

[16] Sakkis, G., Androutsopoulos, I., Paliouras, G., Karkaletsis, V., Spyropoulos, C., and Stamatopoulos, P. A Memory-Based Approach to Anti-Spam Filtering for Mailing Lists. *Information Retrieval Journal 6*, 1 (2003). Available: http://www.eden.rutgers.edu/∼gsakkis/docs/IR2003.pdf.

[17] SpamBayes Development Team. Spambayes. Available: http://spambayes.sourceforge.net.

[18] The Apache SpamAssassin Project. SpamAssassin Public Mail Corpus. Available: http://spamassassin.apache.org/publiccorpus/.

[19] Wittel, G. L., and Wu, S. F. On Attacking Statistical Spam Filters. In *Proceedings of the First Conference on Email and Anti-Spam* (2004). Available: http://www.ceas.cc/papers-2004/170.pdf.

[20] Yerazunis, B. The Plateau at 99.9% Accuracy, and How to Get Past It. In *Proceedings of the Spam Conference* (2004). Available: http://crm114.sourceforge.net/Plateau_Paper.pdf.

[21] Zdziarski, J. A. *Ending Spam: Bayesian Content Filtering and The Art of Statistical Language Classification*. No Starch Press, San Francisco, CA, USA, 2005.

[22] Zhang, L., Zhu, J., and Yao, T. An Evaluation of Statistical Spam Filtering Techniques. *ACM Transactions on Asian Language Information Processing 3*, 4 (Dec. 2004), 243–269.

Appendix A

Source Code

Code from SpamProbe 1.0a is used for tokenization. The following files are used:

AbstractMessageFactory.h, AbstractPhraseBuilder.h, Message.cc, Message.h, MessageFactory.cc, MessageFactory.h, MimeHeader.cc, MimeHeader.h, MimeLineReader.cc, MimeLineReader.h, MimeMessageReader.cc MimeMessage.h, NewPtr.h, PhraseBuilder.h ProximityPhraseBuilder.h, RegularExpression.cc, RegularExpression.h, Token.h, Tokenizer.cc, Tokenizer.h, util.cc, util.h.

For clarity, all SpamProbe related code was modified to be encapsulated in its own namespace. All original code is listed below.

```
//
//
// spamFilter test system
//  version 0.8
//  last updated: Mar03,2006
//
//
// main.cpp
// starting point for program,
// handles command line arguments
//

#include <fstream>
#include <iomanip>
#include <vector>
#include <string>
#include "SpamFilter.hpp"
#include "Message.hpp"
#include "IndexMachine.hpp"
#include "TestingCenter.hpp"
using namespace std;

// boost library used for handling command line arguments
#include <boost/program_options.hpp>
namespace po = boost::program_options;
using namespace boost;

// the possible commands a user can request
enum COMMANDS
{
   COMMAND_TOKENIZE,
   COMMAND_CREATE_INDEXES,
   COMMAND_RUN_TEST
};
```

```
36 const double versionNum = 0.8;

38 int main( int argc, char **argv )
   {
40   COMMANDS commandRequested;
     int numCommandsRequested = 0;
42   Message::MSG_TYPE msgGoldStd = Message::HAM;
     string importFileName = "";
44
     SpamFilter sf;
46   IndexMachine indexMaker;
     TestingCenter tc;
48
     try {
50     // set up the command line arguments
       po::options_description genericOptions( "Generic options" );
52     genericOptions.add_options()
         ( "help", "produce help message" )
54       ( "version,v", "print version string" )
         ;
56
       po::options_description tokenizerOptions( "Tokenizer options" );
58     tokenizerOptions.add_options()
         ( "no-body,b", po::value<bool>(),
60         "set ignore body of messages (false)" )
         ( "no-html,H", po::value<bool>(),
62         "set ignore html tags (true)" )
         ( "headers,h", po::value<string>(),
64         "set headers to include: \n ALL, NONE, NOX, or NORMAL (ALL)" )
         ( "mark-headers,m", po::value<bool>(),
66         "set mark header data (false)" )
         ( "min-phrase-length,p", po::value<int>(),
68         "set minimum phrase length (1)" )
         ( "max-phrase-length,P", po::value<int>(),
70         "set maximum phrase length (1)" )
         ( "tokenize", po::value<string>(),
72         "tokenize given input file" )
         ;
74
       po::options_description trainOptions( "Training options" );
76     trainOptions.add_options()
         ( "delay,d", po::value<int>(),
78         "correctional delay for TEFT-C and TOE\n (in number of errors before
             correction) (1)" )
         ( "train-mode,M", po::value<string>(),
80         "training mode: TEFT, TEFT-C, TOE, or NONE, (TEFT-C)" )
         ;
82
       po::options_description classifyOptions( "Classification options" );
84     classifyOptions.add_options()
         ( "count,c", po::value<int>(),
86         "set minimum count of token \n to allow its usage in decision matrix
             (5)" )
         ( "double,2", po::value<bool>(),
88         "set Graham-style double ham (false)" )
         ( "threshold,T", po::value<double>(),
90         "set decision threshold (0.7)" )
         ( "force,f", po::value<bool>(),
92         "force allow interesting tokens \n in decision matrix (false)" )
```

```
    ( "comb-prob,C", po::value<string>(),
94    "set␣combination␣function:␣\n␣␣graham,␣geo_mean,␣sp_graham␣␣(sp_graham
        )" )
    ( "max-token-score", po::value<double>(),
96    "set␣token␣maximum␣score␣␣(0.999999)␣\n␣␣(applies␣only␣to␣Graham␣token
        ␣prob␣func)" )
    ( "min-token-score", po::value<double>(),
98    "set␣token␣minimum␣score␣␣(0.000001)␣\n␣␣(applies␣only␣to␣Graham␣token
        ␣prob␣func)" )
    ( "new,N", po::value<double>(),
100   "set␣probability␣assigned␣to␣new␣tokens␣␣(0.4)" )
    ( "usage-count,u", po::value<int>(),
102   "set␣number␣of␣times␣a␣\n␣␣token␣can␣be␣used␣in␣decision␣matrix␣␣(1)"
        )
    ( "size,s", po::value<int>(),
104   "set␣minimum␣size␣of␣decision␣matrix␣␣(15)" )
    ;
106
  po::options_description testingOptions( "Testing␣Options" );
108 testingOptions.add_options()
    ( "input-ham", po::value< vector<string> >(),
110   "specify␣ham␣folder␣used␣to␣create␣index" )
    ( "input-spam", po::value< vector<string> >(),
112   "specify␣spam␣folder␣used␣to␣create␣index" )
    ( "create-indexes", po::value<int>(),
114   "create␣specified␣number␣of␣index␣files␣\n␣␣from␣given␣ham␣and␣spam␣
        folders" )
    ( "initial-train-count", po::value<int>(),
116   "set␣number␣of␣messages␣\n␣␣used␣for␣initial␣training␣(0)" )
    ( "id", po::value<string>(),
118   "set␣ID␣of␣this␣test" )
    ( "verbose", po::value<int>(),
120   "set␣verbosity␣level␣(0)␣\n␣␣0:␣normal␣\n␣␣1:␣output␣decision␣matrix␣
        file␣for␣each␣run␣\n␣␣2:␣also␣output␣database␣for␣each␣run" )
    ( "importDB", po::value<string>(),
122   "import␣given␣database␣to␣be␣used␣in␣tests" )
    ( "run-test", po::value<string>(),
124   "run␣test␣on␣given␣folder␣of␣index␣files" )
    ;
126
  po::options_description experimentalOptions( "Experimental␣options" );
128 experimentalOptions.add_options()
    ( "token-prob", po::value<string>(),
130   "set␣token␣probability␣function:␣\n␣␣graham,␣weighted␣␣(weighted)" )
    ( "h-weight", po::value<double>(),
132   "weight␣applied␣to␣header␣tokens␣(1)␣\n␣␣(applies␣only␣to␣weighted␣
        token␣prob␣func)" )
    ( "p-weight", po::value<double>(),
134   "weight␣applied␣to␣multi-word␣tokens␣(1)␣\n␣␣(applies␣only␣to␣weighted
        ␣token␣prob␣func)" )
    ( "weighted-eps", po::value<double>(),
136   "set␣'eps'␣value␣in␣weighted␣token␣prob␣func␣(1)" )
    ( "h-usage-count", po::value<int>(),
138   "set␣number␣of␣times␣a␣header␣token␣can␣\n␣␣be␣used␣in␣decision␣matrix
        ␣␣(1)" )
    ( "p-usage-count", po::value<int>(),
140   "set␣number␣of␣times␣a␣multi-word␣token␣can␣\n␣␣be␣used␣in␣decision␣
        matrix␣␣(1)" )
    ;
```

```
142
    po::options_description cmdline_options( "Allowed␣Options" );
144 cmdline_options.add( genericOptions );
    cmdline_options.add( tokenizerOptions );
146 cmdline_options.add( trainOptions );
    cmdline_options.add( classifyOptions );
148 cmdline_options.add( testingOptions );
    cmdline_options.add( experimentalOptions );
150
    po::variables_map vm;
152 po::store(po::parse_command_line(argc,argv, cmdline_options), vm);
    po::notify( vm );
154

156 ////////////////////////////////////////////////////////////////
    //
158 //   General options
    //
160 //
    if( vm.count( "help" ) )
162 {
      cout << cmdline_options << endl;
164   return 1;
    }
166 if( vm.count( "version" ) )
    {
168   cout << endl << "Spam␣Filter␣Test␣System"
           << endl << "␣␣␣␣␣Version␣" << versionNum << endl;
170   exit(1);
    }
172
    ////////////////////////////////////////////////////////////////
174 //
    //   Tokenize options
176 //
    //
178 if( vm.count( "no-body" ) )
    {
180   cout << "set␣ignore␣body:␣"
        << vm["no-body"].as<bool>() << endl;
182   sf.setIgnoreBody( vm["no-body"].as<bool>() );
    }
184 if( vm.count( "no-html" ) )
    {
186   cout << "set␣ignore␣html:␣"
        << vm["no-html"].as<bool>() << endl;
188   sf.setIgnoreHTML( vm["no-html"].as<bool>());
    }
190 if( vm.count( "headers" ) )
    {
192   cout << "set␣headers:␣"
        << vm["headers"].as<string>() << endl;
194
      sf.setHeadersToInclude( vm["headers"].as<string>() );
196 }
    if( vm.count( "mark-headers" ) )
198 {
      cout << "set␣mark␣headers:␣"
200     << vm["mark-headers"].as<bool>() << endl;
```

```
202       sf.setMarkHeaders( vm["mark-headers"].as<bool>() );
        }
204     if( vm.count( "min-phrase-length" ) && vm.count( "max-phrase-length" ) )
        {
206       if( vm["min-phrase-length"].as<int>() >
            vm["max-phrase-length"].as<int>() )
208       {
            cout << endl << "min-phrase-length␣should␣be␣LESS␣than␣max-phrase-
                length" << endl;
210         exit(1);
          }
212
          cout << "minimum␣phrase␣length␣was␣set␣to:␣"
214         << vm["min-phrase-length"].as<int>() << endl;

216       cout << "maximum␣phrase␣length␣was␣set␣to:␣"
            << vm["max-phrase-length"].as<int>() << endl;
218
          sf.setMinPhraseLength( vm["min-phrase-length"].as<int>() );
220       sf.setMaxPhraseLength( vm["max-phrase-length"].as<int>() );
        }
222     else if( vm.count( "min-phrase-length" ) )
        {
224       if( vm["min-phrase-length"].as<int>() > 1 )
          {
226         cout << endl << "invalid␣min-phrase-length" << endl;
            exit(1);
228       }
          cout << "minimum␣phrase␣length␣was␣set␣to:␣"
230         << vm["min-phrase-length"].as<int>() << endl;

232       sf.setMinPhraseLength( vm["min-phrase-length"].as<int>() );
        }
234     else if( vm.count( "max-phrase-length" ) )
        {
236       if( vm["max-phrase-length"].as<int>() < 1 )
          {
238         cout << endl << "invalid␣max-phrase-length" << endl;
            exit(1);
240       }
          cout << "maximum␣phrase␣length␣was␣set␣to:␣"
242         << vm["max-phrase-length"].as<int>() << endl;

244       sf.setMaxPhraseLength( vm["max-phrase-length"].as<int>() );
        }
246     if( vm.count( "tokenize" ) )
        {
248       cout << "tokenize␣was␣requested" << endl;
          ++numCommandsRequested;
250       commandRequested = COMMAND_TOKENIZE;
          importFileName = vm["tokenize"].as<string>();
252     }

254     ////////////////////////////////////////////////////////////////
        //
256     //   Train options
        //
258     //
```

```
    if( vm.count( "delay" ) )
260 {
      cout << "Delay was set to: " << vm["delay"].as<int>() << endl;
262   sf.setCorrectionDelay( vm["delay"].as<int>() );
    }
264 if( vm.count( "train-mode" ) )
    {
266   cout << "Training mode was set to: " << vm["train-mode"].as<string>()
          << endl;
      sf.setTrainMode( vm["train-mode"].as<string>() );
268 }

270 ////////////////////////////////////////////////////////////////
    //
272 //    Classification options
    //
274 //
    if( vm.count( "count" ) )
276 {
      cout << "set minimum count of token for usage in decision matrix: "
278      << vm["count"].as<int>() << endl;

280   sf.setMinPrevSightings( vm["count"].as<int>() );
    }
282 if( vm.count( "double" ) )
    {
284   cout << "set double ham count: "
        << vm["double"].as<bool>() << endl;
286
      sf.setDoubleHamCount( vm["double"].as<bool>() );
288 }
    if( vm.count( "threshold" ) )
290 {
      cout << "set threshold of spam decision: "
292     << vm["threshold"].as<double>() << endl;

294   sf.setDecisionThreshold( vm["threshold"].as<double>() );
    }
296 if( vm.count( "force" ) )
    {
298   cout << "set force allow interesting tokens: "
        << vm["force"].as<bool>() << endl;
300
      sf.setForceInterestingTokens( vm["force"].as<bool>() );
302 }
    if( vm.count( "comb-prob" ) )
304 {
      cout << "set combined-probability function: "
306     << vm["comb-prob"].as<string>() << endl;

308   sf.setCombProbFunc( vm["comb-prob"].as<string>() );
    }
310 if( vm.count( "max-token-score" ) )
    {
312   cout << "set token maximum score: "
        << vm["max-token-score"].as<double>() << endl;
314
      sf.setTokenMaxScore( vm["max-token-score"].as<double>() );
316 }
```

```
    if( vm.count( "min-token-score" ) )
318 {
      cout << "set token minimum score: "
320     << vm["min-token-score"].as<double>() << endl;

322   sf.setTokenMinScore( vm["min-token-score"].as<double>() );
    }
324 if( vm.count( "new" ) )
    {
326   cout << "set probability of new tokens: "
        << vm["new"].as<double>() << endl;
328
      sf.setTokenHapaxScore( vm["new"].as<double>() );
330 }
    if( vm.count( "usage-count" ) )
332 {
      cout << "set number of time a token can be used in decision matrix: "
334     << vm["usage-count"].as<int>() << endl;

336   sf.setTokenUsageCount( vm["usage-count"].as<int>() );
    }
338 if( vm.count( "size" ) )
    {
340   cout << "set decision matrix minimum size: "
        << vm["size"].as<int>() << endl;
342
      sf.setMatrixMinSize( vm["size"].as<int>() );
344 }

346 ////////////////////////////////////////////////////////////////
    //
348 //    Testing options
    //
350 //
    if( vm.count( "input-ham" ) )
352 {
      cout << "added ham input sources: " << endl;
354   vector<string> hamSources = vm["input-ham"].as<vector<string> >();
      for( size_t i=0; i<hamSources.size(); ++i )
356   {
        cout << hamSources[i] << endl;
358     indexMaker.addSource( hamSources[i], Message::HAM );
      }
360 }
    if( vm.count( "input-spam" ) )
362 {
      cout << "added spam input Sources: " << endl;
364   vector<string> spamSources = vm["input-spam"].as<vector<string> >();
      for( size_t i=0; i<spamSources.size(); ++i )
366   {
        cout << spamSources[i] << endl;
368     indexMaker.addSource( spamSources[i], Message::SPAM );
      }
370 }
    if( vm.count( "create-indexes" ) )
372 {
      ++numCommandsRequested;
374   commandRequested = COMMAND_CREATE_INDEXES;
```

```
376       indexMaker.setNumIndexes( vm["create-indexes"].as<int>() );
        }
378     if( vm.count( "initial-train-count" ) )
        {
380       cout << "set initial training count: "
            << vm["initial-train-count"].as<int>() << endl;
382
          tc.setInitialTrainingCount( vm["initial-train-count"].as<int>() );
384     }
        if( vm.count( "id" ) )
386     {
          cout << "set test id: "
388         << vm["id"].as<string>() << endl;

390       tc.setID( vm["id"].as<string>() );
        }
392     if( vm.count( "verbose" ) )
        {
394       cout << "verbosity level set: " << vm["verbose"].as<int>() << endl;

396       tc.setVerbose( vm["verbose"].as<int>() );
        }
398     if( vm.count( "importDB" ) )
        {
400       cout << "import database: " << vm["importDB"].as<string>() << endl;

402       importFileName = vm["importDB"].as<string>();
        }
404     if( vm.count( "run-test" ) )
        {
406       ++numCommandsRequested;
          commandRequested = COMMAND_RUN_TEST;
408
          if( !tc.setTestSuitePath( vm["run-test"].as<string>() ) )
410         return 1;
        }
412
        ////////////////////////////////////////////////////////////////
        //
        //   Experimental options
        //
        //
418     if( vm.count( "token-prob" ) )
        {
420       cout << "set token probability function: " <<
            vm["token-prob"].as<string>() << endl;
422
          sf.setTokenProbFunc( vm["token-prob"].as<string>() );
424     }
        if( vm.count( "h-weight" ) )
426     {
          cout << "set header token weight: "
428         << vm["h-weight"].as<double>() << endl;

430       sf.setHeaderWeight( vm["h-weight"].as<double>() );
        }
432     if( vm.count( "p-weight" ) )
        {
434       cout << "set multi-word token Weight: "
```

```
        << vm["p-weight"].as<double>() << endl;

      sf.setPhraseWeight( vm["p-weight"].as<double>() );
    }
    if( vm.count( "weighted-eps" ) )
    {
      cout << "set 'eps' value in weighted token prob func: "
        << vm["weighted-eps"].as<double>() << endl;

      sf.setTPFWeightedEps( vm["weighted-eps"].as<double>() );
    }
    if( vm.count( "h-usage-count" ) )
    {
      cout << "set header token usage count: "
        << vm["h-usage-count"].as<int>() << endl;

      sf.setHTokenUsageCount( vm["h-usage-count"].as<int>() );
    }
    if( vm.count( "p-usage-count" ) )
    {
      cout << "set multi-word token usage count: "
        << vm["p-usage-count"].as<int>() << endl;

      sf.setPTokenUsageCount( vm["p-usage-count"].as<int>() );
    }
  }
  catch( exception &e )
  {
    cout << "error: " << e.what() << endl;
  }


  // did the user issue some command?
  if( numCommandsRequested == 0 )
  {
    cout << "Error: No Commands Specified" << endl;
    cout << "  Use --help for allowed options" << endl;
    return -1;
  }
  // only one command can be served at a time
  else if( numCommandsRequested > 1 )
  {
    cout << "Error: Multiple Commands Specified" << endl;
    return -1;
  }
  else
  {
    // just tokenize the given message
    if( commandRequested == COMMAND_TOKENIZE )
    {
      ifstream inFile;
      inFile.open( importFileName.c_str() );
      if( !inFile )
      {
        cout << "Could not open file: " << importFileName << endl;
        exit(1);
      }
      Message msg;
      sf.tokenize( inFile, msg );
```

```
494         inFile.close();

496         msg.printShort( cout );
          }
498       // create indexes was requested
          else if( commandRequested == COMMAND_CREATE_INDEXES )
500       {
            indexMaker.createIndexes();
502       }
          // user wants to run a test
504       else if( commandRequested == COMMAND_RUN_TEST )
          {
506         // did the user supply a DB?
            //  if so, open it
508         if( importFileName != "" )
              sf.openDB( importFileName );
510
            // connect the spamfilter to the testing center,
512         //  then run the test
            tc.setSpamFilter( &sf );
514         tc.runTests();
          }
516     }

518     return 0;
    }


    //
  2 // SpamFilter.hpp
    //
  4 // class interface
    //
  6 // this class represents the heart of the spam filter
    // it scores tokens, messages, and trains the database
  8 //

 10 #pragma once

 12 #include "SpamProbeTokenizer.hpp"
    #include "Message.hpp"
 14 #include "Token.hpp"
    #include "TokenDB_map.hpp"
 16 #include "TokenDB_hashmap.hpp"
    #include "TrainStation.hpp"
 18 #include "DecisionMatrixFactory.hpp"
    using namespace std;
 20
    class SpamFilter
 22 {
    public:
 24   SpamFilter(void);
      ~SpamFilter(void);
 26
      enum COMB_PROB_FUNC
 28   {
        CPF_GRAHAM,
 30     CPF_GEO_MEAN,
        CPF_SP_GRAHAM
 32   };

 34   enum TOKEN_PROB_FUNC
      {
 36     TPF_GRAHAM,
        TPF_WEIGHTED
```

```
38    };
40    void tokenize(
        istream &in,
42      Message &msg );
44    void train(
        istream &in,
46      Message::MSG_TYPE goldStd,
        Message::MSG_TYPE prevDec );
48
      void initialTrain(
50      istream &in,
        Message::MSG_TYPE goldStd );
52
      void classify(
54      istream &in,
        Message::MSG_TYPE &decision,
56      double &score,
        int verbose,
58      ostream &out );
60    void openDB( const string &fileName );
      void resetDB(void);
62    int getDBTokenCount(void) const;
64    void printDB( ostream &out );
      void setMinPhraseLength( int value );
66    void setMaxPhraseLength( int value );
      void setIgnoreBody( bool value );
68    void setIgnoreHTML( bool value );
      void setMarkHeaders( bool value );
70    void setHeadersToInclude( string value );
      void setCorrectionDelay( int delay );
72    void setTrainMode( string mode );
      void setMinPrevSightings( int count );
74    void setDecisionThreshold( double threshold );
      void setForceInterestingTokens( bool value );
76    void setTokenUsageCount( int num );
      void setMatrixMinSize( int num );
78    void setDoubleHamCount( bool value );
      void setTokenHapaxScore( double value );
80    void setTokenMinScore( double value );
      void setTokenMaxScore( double value );
82    void setCombProbFunc( string mode );
      void setTokenProbFunc( string mode );
84    void setHeaderWeight( double value );
      void setPhraseWeight( double value );
86    void setTPFWeightedEps( double value );
      void setHTokenUsageCount( int value );
88    void setPTokenUsageCount( int value );
90  private:
      void scoreMessageTokens( Message &msg ) const;
92    double tokenProbGraham( Token *tok ) const;
      double tokenProbWeighted( Token *tok ) const;
94    double combProbGraham( vector<Token*> decisionMatrix ) const;
      double combProbGeoMean( vector<Token*> decisionMatrix ) const;
96    double combProbSPGraham( vector<Token*> decisionMatrix ) const;
      double constrainScore( double score ) const;
98
    protected:
100   SpamProbeTokenizer m_tokenizer;
      TrainStation m_trainer;
102   string m_dbFileName;
      TokenDB *m_db;
104   DecisionMatrixFactory m_matrixFactory;
      bool m_doubleHamCount;
```

```
106   double m_tokenHapaxScore;
      double m_tokenMinScore;
108   double m_tokenMaxScore;
      double m_decisionThreshold;
110   COMB_PROB_FUNC m_combProbFunc;
      TOKEN_PROB_FUNC m_tokenProbFunc;
112   double m_headerWeight;
      double m_phraseWeight;
114   double m_TPFWeightedEps;
  };

  //
2 // SpamFilter.cpp
  //
4 // class implementation
  //
6 // this class represents the heart of the spam filter
  // it scores tokens, messages, and trains the database
8 //

10 #include "SpamFilter.hpp"

12 // default constructor
  SpamFilter::SpamFilter(void)
14 : m_doubleHamCount( false ),
    m_tokenHapaxScore( 0.4 ),
16  m_tokenMinScore( 0.000001 ),
    m_tokenMaxScore( 0.999999 ),
18  m_decisionThreshold( 0.7 ),
    m_combProbFunc( CPF_SP_GRAHAM ),
20  m_tokenProbFunc( TPF_WEIGHTED ),
    m_headerWeight( 1 ),
22  m_phraseWeight( 1 ),
    m_TPFWeightedEps( 1 ),
24  m_dbFileName( "" )
  {
26  // currently using the hashmap DB
    m_db = new TokenDB_hashmap();
28 }

30 // destructor
  // close the DB, then delete
32 SpamFilter::~SpamFilter(void)
  {
34  m_db->close();
    delete m_db;
36 }

38 void SpamFilter::openDB( const string &fileName )
  {
40  m_dbFileName = fileName;
    m_db->open( fileName );
42 }

44 void SpamFilter::resetDB(void)
  {
46  m_db->close();

48  if( m_dbFileName != "" )
      m_db->open( m_dbFileName );
50 }

52 int SpamFilter::getDBTokenCount(void) const
  {
54  return m_db->getDBTokenCount();
  }
56
  ////////////////////////////////////////////
```

```
58 //
   //  Tokenize the given input stream,
60 //   building the given message
   //
62 void SpamFilter::tokenize(
     istream &in,
64   Message &msg )
   {
66   m_tokenizer.tokenize( in, msg );
   }
68
   ////////////////////////////////////////////////
70 //
   //  take the given input stream,
72 //  tokenize it,
   //  build a message,
74 //  hand the training the message and database,
   //   it will add the message's tokens to the database
76 //   as needed in the currect training scheme
   //
78 void SpamFilter::train(
     istream &in,
80   Message::MSG_TYPE goldStd,
     Message::MSG_TYPE prevDec )
82 {
     Message msg;
84   this->tokenize( in, msg );

86   m_trainer.train( msg, goldStd, prevDec, *m_db );
   }
88
   ////////////////////////////////////////////////
90 //
   //  during initial training phase,
92 //  take the given input stream,
   //  tokenize it,
94 //  build a message,
   //  hand the trainer the message and database,
96 //   it will apply the message's tokens to the database
   //
98 //  during initial training phase,
   //  all tokens are added to the database
100 //
   void SpamFilter::initialTrain(
102   istream &in,
     Message::MSG_TYPE goldStd )
104 {
     Message msg;
106  this->tokenize( in, msg );

108  m_trainer.initialTrain( msg, goldStd, *m_db );
   }
110
   void SpamFilter::printDB( ostream &out )
112 {
     m_db->print( out );
114 }

116 ////////////////////////////////////////////////
   //
118 //  tokenize the given input stream,
   //  build a message,
120 //  score the tokens in the message,
   //  compute an overall score.
122 //  return the score and decision
   //
124 void SpamFilter::classify(
     istream &in,
```

```
126     Message::MSG_TYPE &decision,
        double &score,
128     int verbose,
        ostream &out )
130 {

132     // tokenize the message, then score the individual tokens
        Message msg;
134     this->tokenize( in, msg );
        this->scoreMessageTokens( msg );
136
        vector<Token*> decisionMatrix;
138     m_matrixFactory.buildDecisionMatrix( msg, decisionMatrix );

140     // which combined probability function are we using?
        //  score the message
142     switch( m_combProbFunc )
        {
144     case CPF_GRAHAM:
          score = combProbGraham( decisionMatrix ); break;
146     case CPF_GEO_MEAN:
          score = combProbGeoMean( decisionMatrix ); break;
148     case CPF_SP_GRAHAM:
          score = combProbSPGraham( decisionMatrix ); break;
150     }

152     // so is the score hammy or spammy?
        if( score >= m_decisionThreshold )
154       decision = Message::SPAM;
        else
156       decision = Message::HAM;

158     // is verbosity on?
        //  if so, output decision matrix to output stream
160     if( verbose >= 1 )
        {
162
          int precisionSetting = out.precision();
164       long flagSettings = out.flags();

166       out.setf( ios::fixed | ios::showpoint | ios::left );
          out.precision( 6 );
168
          out << "␣" << score << "␣";
170
          switch( decision )
172       {
          case Message::SPAM: out << "SPAM" << endl; break;
174       case Message::HAM: out << "HAM" << endl; break;
          };
176
          out << setw(6) << "Count"
178         << setw(6) << "Ham"
            << setw(6) << "Spam"
180         << setw(10) << "Score"
            << "Token" << endl;
182
          for( size_t i=0; i<decisionMatrix.size(); ++i )
184       {
            out << setw(6) << decisionMatrix[i]->getCount()
186           << setw(6) << decisionMatrix[i]->getHamCount()
              << setw(6) << decisionMatrix[i]->getSpamCount()
188           << setw(10) << decisionMatrix[i]->getScore()
              << decisionMatrix[i]->getTok() << endl;
190       }

192       out << endl;
```

```
194       out.precision( precisionSetting );
          out.flags( flagSettings );
196     }
    }
198
    ////////////////////////////////////////////
200 //
    //  loop through tokens in message,
202 //  score each
    //
204 void SpamFilter::scoreMessageTokens( Message &msg ) const
    {
206   int hamCount = 0;
      int spamCount = 0;
208   double score = 0;

210   for( int i=0; i<msg.getNumTokens(); ++i )
      {
212     Token *currTok = msg.getToken( i );

214     // get token's counts from the database
        m_db->getTokenCounts( currTok->getTok(), hamCount, spamCount );
216     currTok->setHamCount( hamCount );
        currTok->setSpamCount( spamCount );
218
        // which token probability function are we using?
220     switch( m_tokenProbFunc )
        {
222     case TPF_GRAHAM:
          score = tokenProbGraham( currTok ); break;
224     case TPF_WEIGHTED:
          score = tokenProbWeighted( currTok ); break;
226     }

228     currTok->setScore( score );
      }
230 }

232
    ////////////////////////////////////////////////////////
234 //
    //  the original token probability function from Graham
236 //
    // g(w) = hamCount / numHamMsgs
238 // b(w) = spamCount / numSpamMsgs
    // p(w) = b(w) / (b(w)+g(w))
240 //
    // graham also double the hamCount...
242 //  which I have as optional
    //
244 // p(w) is limited to a specified min and max
    //
246 double SpamFilter::tokenProbGraham( Token *tok ) const
    {
248   double score = 0;
      double g = 0;
250   double b = 0;
      int numHamMsgs = 0;
252   int numSpamMsgs = 0;

254   // get message counts from database
      m_db->getTokenCounts( TrainStation::MESSAGE_COUNTER, numHamMsgs, numSpamMsgs);
256
      int spamCount = tok->getSpamCount();
258   int hamCount = tok->getHamCount();

260   // never seen this token before
      if( hamCount == 0 && spamCount == 0 )
```

```
262       return m_tokenHapaxScore;

264     // haven't seen this token before in ham,
        //  so it *must* be very spammy
266     if( hamCount == 0 )
          return m_tokenMaxScore;
268
        // haven't seen this token before in spam,
270     //  so it *must* be very hammy
        if( spamCount == 0 )
272       return m_tokenMinScore;

274     // is the graham-like hammy fudge factor turned on?
        if( m_doubleHamCount )
276       hamCount *= 2;

278     // when you haven't processed any ham or spam yet,
        //  default them to 1 (to avoid a DIVBYZERO)
280     // (rare, since you only score tokens after the initial training phase,
        //  unless you're trying to score with an absolutely empty DB,
282     //  which only I would be crazy enough to try)
        numHamMsgs = max( numHamMsgs, 1 );
284     numSpamMsgs = max( numSpamMsgs, 1 );

286     b = static_cast<double>(spamCount) /
          static_cast<double>(numSpamMsgs);
288
        g = static_cast<double>(hamCount) /
290       static_cast<double>(numHamMsgs);

292     score = (b / (b+g));

294     // apply limits to the score
        score = constrainScore( score );
296     return score;
    }
298
    ////////////////////////////////////////////////////////
300 //
    //  our modified token probability function
302 //
    //  added an 'eps' value to eliminate hard limites
304 //  added weights for header and phrased tokens
    //
306 //  g(w) = (weight * hamCount + eps ) / (numHamMsgs + eps )
    //  b(w) = (weight * spamCount + eps ) / numSpamMsgs + eps )
308 //  p(w) = b(w) / (b(w) + g(w))
    //
310 double SpamFilter::tokenProbWeighted( Token *tok ) const
    {
312   double score = 0;
      double g = 0;
314   double b = 0;
      int numHamMsgs = 0;
316   int numSpamMsgs = 0;
      double weight = 1;
318
      // get the message counts from the database
320   m_db->getTokenCounts( TrainStation::MESSAGE_COUNTER, numHamMsgs, numSpamMsgs);

322   double spamCount = tok->getSpamCount();
      double hamCount = tok->getHamCount();
324
      // never seen this token before
326   if( hamCount == 0 && spamCount == 0 )
        return m_tokenHapaxScore;
328
      // build the 'weight'
```

```
330   if( tok->isHeaderToken() )
        weight *= m_headerWeight;
332   if( tok->isPhraseToken() )
        weight *= m_phraseWeight;
334
      // is the graham-like hammy fudge factor turned on?
336   if( m_doubleHamCount )
        hamCount *= 2;
338
      // apply weights
340   hamCount *= weight;
      spamCount *= weight;
342
      // add 'eps' value to token counts
344   hamCount += m_TPFWeightedEps;
      spamCount += m_TPFWeightedEps;
346
      // calculate 'b' and 'g', with 'eps' added to message counts
348
      b = spamCount /
350     (static_cast<double>(numSpamMsgs) + m_TPFWeightedEps );
      g = hamCount /
352     (static_cast<double>(numHamMsgs) + m_TPFWeightedEps );

354   score = (b / (b+g));
      return score;
356 }

358 ////////////////////////////////////////////////////////
    //
360 //  Graham's original combined probability function
    //
362 //  P = s / s + g
    //    where s = x1 * x2 * x3 * ... * xn
364 //          g = (1-x1)*(1-x2)*...*(1-xn)
    //
366 //
    double SpamFilter::combProbGraham( vector<Token*> decisionMatrix ) const
368 {
      double score = 0.0;
370   double s = 1.0;
      double g = 1.0;
372
      if( decisionMatrix.size() == 0 )
374     return m_tokenHapaxScore;

376   // build products
      for( size_t i=0; i<decisionMatrix.size(); ++i )
378   {
        s = s * decisionMatrix[i]->getScore();
380     g = g * (1.0 - decisionMatrix[i]->getScore() );
      }
382
      score = s / (s+g);
384   return score;
    }
386

388 ////////////////////////////////////////////////////////
    //
390 //  Gary Robinson's Geometric Mean
    //    combined probability function
392 //
    //  P = 1 - pow( ((1-p1)*(1-p2)*...*(1-pn)), (1/n) )
394 //  Q = 1 - pow( ((p1)*(p2)*...*(pn)), (1/n) )
    //  S = (1 + (P-Q)/(P+Q)) / 2
396 //
    double SpamFilter::combProbGeoMean( vector<Token*> decisionMatrix ) const
```

```
398 {
    double score = 0.0;
400   double p = 1.0;
    double q = 1.0;
402   double s = 1.0;
    double g = 1.0;
404
    int n = static_cast<int>(decisionMatrix.size());
406   if( n == 0 )
      return m_tokenHapaxScore;
408
    for( size_t i=0; i<decisionMatrix.size(); ++i )
410   {
      s = s * decisionMatrix[i]->getScore();
412     g = g * (1.0 - decisionMatrix[i]->getScore() );
    }
414
    p = 1.0 - pow( g, (1.0/n) );
416   q = 1.0 - pow( s, (1.0/n) );

418   score = ( p - q ) / ( p + q );
    score = ( score + 1.0 ) / 2.0;
420
    return score;
422 }

424 ////////////////////////////////////////////////
  //
426 //  Modified Graham-like combined probability function
  //  created by the SpamProbe project
428 //  http://spamprobe.sourceforge.net
  //
430 double SpamFilter::combProbSPGraham( vector<Token*> decisionMatrix ) const
  {
432   double score = 0;
    double s = 1.0;
434   double g = 1.0;

436   int n = static_cast<int>(decisionMatrix.size());
    if( n == 0 )
438     return m_tokenHapaxScore;

440   for( size_t i=0; i<decisionMatrix.size(); ++i )
    {
442     s = s * decisionMatrix[i]->getScore();
      g = g * (1.0 - decisionMatrix[i]->getScore());
444   }

446   s = pow( s, (1.0 / n) );
    g = pow( g, (1.0 / n) );
448
    score = s / (s+g);
450   return score;
  }
452
  ////////////////////////////////////////////////
454 //
  //  put hard limits on the token probability score
456 //  used by Graham's original token prob func
  //
458 double SpamFilter::constrainScore( double score ) const
  {
460   score = min( m_tokenMaxScore, score );
    score = max( m_tokenMinScore, score );
462   return score;
  }
464
  ////////////////////////////////////////////////////////////
```

```
466 ////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////
468 ////////////////////////////////////////////////////////////
    //
470 //  Short modifiers for tokenizer, trainer, classifier, decision matrix
    //
472 ////////////////////////////////////////////////////////////

474 void SpamFilter::setMinPhraseLength( int value )
    {
476   m_tokenizer.setMinPhraseLength( value );
    }
478
    void SpamFilter::setMaxPhraseLength( int value )
480 {
      m_tokenizer.setMaxPhraseLength( value );
482 }

484 void SpamFilter::setIgnoreBody( bool value )
    {
486   m_tokenizer.setIgnoreBody( value );
    }
488
    void SpamFilter::setIgnoreHTML( bool value )
490 {
      m_tokenizer.setIgnoreHTML( value );
492 }

494 void SpamFilter::setMarkHeaders( bool value )
    {
496   m_tokenizer.setMarkHeaders( value );
    }
498
    void SpamFilter::setHeadersToInclude( string value )
500 {
      m_tokenizer.setHeadersToInclude( value );
502 }

504 void SpamFilter::setCorrectionDelay( int delay )
    {
506   m_trainer.setCorrectionDelay( delay );
    }
508
    void SpamFilter::setTrainMode( string mode )
510 {
      m_trainer.setTrainMode( mode );
512 }

514 void SpamFilter::setMinPrevSightings( int count )
    {
516   m_matrixFactory.setMinPrevSightings( count );
    }
518
    void SpamFilter::setDecisionThreshold( double threshold )
520 {
      m_decisionThreshold = threshold;
522 }

524 void SpamFilter::setForceInterestingTokens( bool value )
    {
526   m_matrixFactory.setForceInteresting( value );
    }
528
    void SpamFilter::setTokenUsageCount( int num )
530 {
      m_matrixFactory.setTokenUsageCount( num );
532 }
```

```
534 void SpamFilter::setMatrixMinSize( int num )
    {
536   m_matrixFactory.setMinMatrixSize( num );
    }
538
    void SpamFilter::setDoubleHamCount( bool value )
540 {
      // this value applies to both the spamFilter
542   //  and in the decision matrix factory
      m_doubleHamCount = value;
544   m_matrixFactory.setDoubleHamCount( value );
    }
546
    void SpamFilter::setTokenHapaxScore( double value )
548 {
      m_tokenHapaxScore = value;
550 }

552 void SpamFilter::setTokenMinScore( double value )
    {
554   m_tokenMinScore = value;
    }
556
    void SpamFilter::setTokenMaxScore( double value )
558 {
      m_tokenMaxScore = value;
560 }

562 void SpamFilter::setCombProbFunc( string mode )
    {
564   transform( mode.begin(), mode.end(), mode.begin(), toupper );

566   if( mode == "GRAHAM" )
        m_combProbFunc = CPF_GRAHAM;
568   else if( mode == "GEO_MEAN" )
        m_combProbFunc = CPF_GEO_MEAN;
570   else if( mode == "SP_GRAHAM" )
        m_combProbFunc = CPF_SP_GRAHAM;
572 }

574 void SpamFilter::setTokenProbFunc( string mode )
    {
576   transform( mode.begin(), mode.end(), mode.begin(), toupper );

578   if( mode == "GRAHAM" )
        m_tokenProbFunc = TPF_GRAHAM;
580   else if( mode == "WEIGHTED" )
        m_tokenProbFunc = TPF_WEIGHTED;
582 }

584 void SpamFilter::setHeaderWeight( double value )
    {
586   m_headerWeight = value;
    }
588
    void SpamFilter::setPhraseWeight( double value )
590 {
      m_phraseWeight = value;
592 }

594 void SpamFilter::setTPFWeightedEps( double value )
    {
596   m_TPFWeightedEps = value;
    }
598
    void SpamFilter::setHTokenUsageCount( int value )
600 {
      m_matrixFactory.setHTokenUsageCount( value );
```

```
602 }

604 void SpamFilter::setPTokenUsageCount( int value )
    {
606   m_matrixFactory.setPTokenUsageCount( value );
    }

    //
  2 // SpamProbeTokenizer.hpp
    //
  4 // class interface
    //
  6 // tokenizer class,
    //  utilizes code from the SpamProbe project
  8 //  http://spamprobe.sourceforge.net
    //
 10 // currently using the tokenizer from SP1.0a
    //
 12
    #pragma once
 14
    #include <iostream>
 16 #include <algorithm>
    #include "SpamProbeTokenizer/MessageFactory.h"
 18 #include "Message.hpp"
    using namespace std;
 20
    class SpamProbeTokenizer
 22 {
    public:
 24   SpamProbeTokenizer(void);
      ~SpamProbeTokenizer(void);
 26
      void tokenize( istream &in, Message &message );
 28   void setMinPhraseLength( int value );
      void setMaxPhraseLength( int value );
 30   void setIgnoreBody( bool value );
      void setIgnoreHTML( bool value );
 32   void setMarkHeaders( bool value );
      void setHeadersToInclude( string value );
 34
    protected:
 36   SpamProbe::MessageFactory m_factory;
      bool m_markHeaders;
 38
    };

    //
  2 // SpamProbeTokenizer.cpp
    //
  4 // class implementation
    //
  6 // tokenizer class,
    //  utilizes code from the SpamProbe project
  8 //  http://spamprobe.sourceforge.net
    //
 10 // currently using the tokenizer from SP1.0a
    //
 12
    #include "SpamProbeTokenizer.hpp"
```

```
14
   // default constructor
16 SpamProbeTokenizer::SpamProbeTokenizer(void)
   {
18   m_factory.setReplaceNonAsciiChars(-1);
     m_factory.setReplaceNonAsciiChars('Z');
20   setMaxPhraseLength( 1 );
     setMinPhraseLength( 1 );
22   setHeadersToInclude( "ALL" );
     setMarkHeaders( false );
24   setIgnoreHTML( true );
   }
26
   // destructor
28 // intentionally empty
   SpamProbeTokenizer::~SpamProbeTokenizer(void)
30 {}

32 // tokenize the input stream,
   //  build a message
34 void SpamProbeTokenizer::tokenize( istream &in, Message &message )
   {
36   bool ignore_from = false;
     bool ignore_content_length = false;
38
     // set up the SP tokenizer
40   SpamProbe::Message msg;
     SpamProbe::MimeMessageReader inReader( in, ignore_from,
         ignore_content_length, !true );
42   msg.setReader(&inReader);
     inReader.readNextHeader();
44   m_factory.initMessage(msg, inReader);

46   // dump from SpamProbe::Message into our Message argument
     for (int i=0; i < msg.getTokenCount(); ++i) {
48     SpamProbe::Token *tok = msg.getToken(i);
       string word = tok->getWord();
50
       // if we're not marking headers,
52     //  we might need to clean the tokens,
       //  since the spamprobe tokenizer always marks
54     if( m_markHeaders == false )
       {
56       // try to find an '_', which means it was marked
         int loc = static_cast<int>(word.find_last_of( '_' ));
58
         if( loc != -1 )  // if there was an '_'
60       {
           // chop off everything before (and including) the '_'
62         word = word.substr( loc + 1 );
         }
64     }

66     message.addToken( word, tok->getCount(), !m_markHeaders );
     }
68 }

70 void SpamProbeTokenizer::setMinPhraseLength( int value )
   {
```

```
72    m_factory.setMinPhraseLength( value );
  }
74
  void SpamProbeTokenizer::setMaxPhraseLength( int value )
76 {
    m_factory.setMaxPhraseLength( value );
78 }

80 void SpamProbeTokenizer::setIgnoreBody( bool value )
  {
82    m_factory.setIgnoreBody( value );
  }
84
  void SpamProbeTokenizer::setIgnoreHTML( bool value )
86 {
    m_factory.setRemoveHTML( value );
88 }

90 void SpamProbeTokenizer::setMarkHeaders( bool value )
  {
92    m_markHeaders = value;
  }
94
  void SpamProbeTokenizer::setHeadersToInclude( string value )
96 {
    transform( value.begin(), value.end(), value.begin(), toupper );
98
    if( value == "ALL" )
100   m_factory.setHeadersToInclude( SpamProbe::MessageFactory::ALL_HEADERS );
    else if( value == "NONE" )
102   m_factory.setHeadersToInclude( SpamProbe::MessageFactory::NO_HEADERS );
    else if( value == "NOX" )
104   m_factory.setHeadersToInclude( SpamProbe::MessageFactory::NO_X_HEADERS );
    else if( value == "NORMAL" )
106 {
      m_factory.addPrefixedHeader("from");
108   m_factory.addPrefixedHeader("to");
      m_factory.addPrefixedHeader("cc");
110   m_factory.addPrefixedHeader("subject");
      m_factory.addPrefixedHeader("received", "recv");
112
      m_factory.setHeadersToInclude( SpamProbe::MessageFactory::NORMAL_HEADERS )
          ;
114 }
  }


  //
2 // Message.hpp
  //
4 // class interface
  //
6 // this class represents a simple message,
  // mainly by the tokens vector
8 //

10 #pragma once

12 #include "Token.hpp"
  #include <vector>
14 #include <iostream>
  #include <iomanip>
```

```
16 #include <algorithm>
   using namespace std;
18
   class Message
20 {
   public:
22   Message(void);
     ~Message(void);
24
     void addToken( const string &word, int count, bool needToCheck = false );
26   Token *getToken( int index ) const;
     int getNumTokens(void) const;
28   void printShort( ostream &out ) const;
     void printAll( ostream &out ) const;
30   void sortMsg(void);

32   // this enum is used by many other classes
     enum MSG_TYPE
34   {
       HAM,
36     SPAM
     };
38
   protected:
40   vector<Token*> m_tokens;
   };


   //
 2 // Message.cpp
   //
 4 // class implementation
   //
 6 // this class represents a simple message,
   // mainly by the tokens vector
 8 //

10 #include "Message.hpp"

12 // default constructor
   // intentionally empty
14 Message::Message(void)
   {}
16
   // destructor
18 // delete all tokens
   Message::~Message(void)
20 {
     for( size_t i=0; i<m_tokens.size(); ++i )
22   {
       delete m_tokens[i];
24   }

26   m_tokens.clear();
   }
28
   // addToken
30 void Message::addToken( const string &word, int count, bool needToCheck )
   {
32   // do we need to check if the token is already in the message?
     if( needToCheck )
34   {
       for( size_t i=0; i<m_tokens.size(); ++i )
36     {
         if( m_tokens[i]->getTok() == word )
38       {
           // we've found the token,
40         // so update its count, then return
           m_tokens[i]->incCount( count );
```

```
42          return;
          }
44      }
    }
46
    // either the token wasn't found,
48  //  or we didn't need to look for it.
    // add it to the message
50  m_tokens.push_back( new Token( word, count ) );
}
52
// getToken
54 // return the requested tokens,
// or NULL if invalid request
56 Token *Message::getToken( int i ) const
{
58  if( i >= static_cast<int>(m_tokens.size())
      || i < 0 )
60  {
      return NULL;
62  }

64  return m_tokens[i];
}
66
int Message::getNumTokens(void) const
68 {
  return static_cast<int>(m_tokens.size());
70 }

72 // output just token string, and count
void Message::printShort( ostream &out ) const
74 {
  for( int i=0; i<static_cast<int>(m_tokens.size()); ++i )
76  {
    out << setw(6) << getToken( i )->getCount()
78      << "  " << getToken( i )->getTok() << endl;
  }
80 }

82 // output all info about a token
void Message::printAll( ostream &out ) const
84 {
  for( int i=0; i<static_cast<int>(m_tokens.size()); ++i )
86  {
    out << setw(6) << getToken( i )->getCount()
88      << setw(6) << getToken( i )->getHamCount()
      << setw(6) << getToken( i )->getSpamCount()
90      << setw(10) << getToken( i )->getScore()
      << "  " << getToken( i )->getTok() << endl;
92  }
}
94
// sort the message,
96 // using the Token::compare() function
// (STL's sort uses introsort. worst case is Nlog(N) )
98 void Message::sortMsg(void)
{
100 sort( m_tokens.begin(), m_tokens.end(), Token::compare );
}


//
2 // Token.hpp
//
4 // class interface
//
6 // a token - its token string, and counts
//
```

```
8
  #pragma once
10
  #include <string>
12 #include <iostream>
  #include <cmath>
14 using namespace std;

16 class Token
  {
18 public:
    Token( const string &tok, unsigned int count );
20  ~Token(void);

22  void setScore( double score );
    double getScore(void) const;
24
    void setTok( const string &tok );
26  string getTok(void) const;

28  void incCount( int change = 1 );
    int getCount(void) const;
30
    void setHamCount( unsigned int count );
32  int getHamCount(void) const;

34  void setSpamCount( unsigned int count );
    int getSpamCount(void) const;
36
    double getDistanceFromMean(void) const;
38
    static bool compare( Token* tok1, Token* tok2 );
40  bool isHeaderToken(void);
    bool isPhraseToken(void);
42
  protected:
44  string m_tok;
    int m_count;
46  int m_hamCount;
    int m_spamCount;
48  double m_score;

50 };
```

```
  //
2 // Token.cpp
  //
4 // class implementation
  //
6 // a token - its token string, and counts
  //
8
  #include "Token.hpp"
10
  // only constructor
12 Token::Token( const string &tok, unsigned int count )
  : m_tok( tok ),
14   m_count( count ),
    m_score( -1 ),
16  m_hamCount( 0 ),
    m_spamCount( 0 )
18 {}

20 // destructor
  // intentionally empty
22 Token::~Token(void)
  {}
24
```

```
  // how far is this token from 0.5?
26 // the score should have been set first
  double Token::getDistanceFromMean(void) const
28 {
    double result = 0;
30   result = 0.5 - m_score;
    return fabs( result );
32 }

34 void Token::setScore( double score )
  {
36   m_score = score;
  }
38
  double Token::getScore(void) const
40 {
    return m_score;
42 }

44 void Token::setTok( const string &tok )
  {
46   m_tok = tok;
  }
48
  string Token::getTok(void) const
50 {
    return m_tok;
52 }

54 void Token::incCount( int change )
  {
56   m_count += change;
  }
58
  int Token::getCount(void) const
60 {
    return m_count;
62 }

64 void Token::setHamCount( unsigned int count )
  {
66   m_hamCount = count;
  }
68
  int Token::getHamCount(void) const
70 {
    return m_hamCount;
72 }

74 void Token::setSpamCount( unsigned int count )
  {
76   m_spamCount = count;
  }
78
  int Token::getSpamCount(void) const
80 {
    return m_spamCount;
82 }

84 // Token comparison function
  // used when sorting a container of tokens
86 // sort first by distance from mean,
  //  if tied favor hammy tokens,
88 //  if tied, favor token with higher count
  //  (does that last comparison matter?)
90 bool Token::compare( Token* tok1, Token* tok2 )
  {
92   // how far is each from the mean probability?
```

```
    double dist1 = tok1->getDistanceFromMean();
    double dist2 = tok2->getDistanceFromMean();

    // sort first by distance from mean
    //  --- farther from mean == more important
    if( dist1 > dist2 )
    {
      return true;
    }
    else if( fabs( dist1 - dist2 ) <= 0.00001 )
    {
      // if distFromMean tie,
      //  check score
      //  use the token with the lower score,
      //  to favor the hammy tokens
      if( tok1->getScore() < tok2->getScore() )
      {
        return true;
      }
      else if( fabs( tok1->getScore() - tok2->getScore() ) <= 0.00001 )
      {
        // if they have they same count,
        //  check count
        //   --- a higher count is considered more important
        if( tok1->getCount() > tok2->getCount() )
        {
          return true;
        }
      }
    }

    return false;
  }

  bool Token::isHeaderToken(void)
  {
    // this is simplified, but works,
    //  since the SpamProbeTokenizer disregards case,
    //  a capital 'H' will only be seen as a header token
    if( m_tok[0] == 'H' )
      return true;

    return false;
  }

  bool Token::isPhraseToken(void)
  {
    // phrased (multi-word) tokens are the only tokens
    //  that may contain a space
    size_t index;
    index = m_tok.find( ' ', 0 );
    if( index != string::npos )
      return true;

    return false;
  }


  //
  // TokenDB.hpp
  //
  // class interface
  //
  // an abstract (pure virtual) base class
  //

  #pragma once

  #include <string>
```

```
#include <iostream>
#include <fstream>
using namespace std;

class TokenDB
{
public:
  TokenDB(void);
  virtual ~TokenDB(void) = 0;

  virtual bool open( const string &fileName = "" ) = 0;
  virtual bool close() = 0;
  virtual bool print( ostream &out ) = 0;

  virtual bool addToken( const string &token, int hamCount, int spamCount ) = 0;
  virtual bool removeToken( const string &token ) = 0;
  virtual bool getTokenCounts( const string &token, int &hamCount, int &spamCount )
      = 0;
  virtual int getDBTokenCount(void) const = 0;
  virtual void mergeDB( TokenDB *db2 ) = 0;

protected:
  string m_fileName;
  virtual void clear(void) = 0;

  class TokenData
  {
  public:
    TokenData( unsigned int hamCount = 0, unsigned int spamCount = 0 )
      : m_hamCount( hamCount ), m_spamCount( spamCount ){};
    unsigned int m_hamCount;
    unsigned int m_spamCount;
  };
};
```

```
//
// TokenDB.cpp
//
// class implementationnterface
//
// an abstract (pure virtual) base class
//

#include "TokenDB.hpp"

// default constructor
TokenDB::TokenDB(void)
: m_fileName("")
{}

// destructor
// intentionally empty
TokenDB::~TokenDB(void)
{}
```

```
//
// TokenDB_hashmap.cpp
//
// class interface
//
// TokenDB using an STL hashmap
//

#pragma once

#include <hash_map>
#include <iomanip>
#include "TokenDB.hpp"
```

```
14 using namespace stdext;

16 class TokenDB_hashmap :
     public TokenDB
18 {
     // class stringhasher
20   //  from codeguru.com
     //
22   // http://www.codeguru.com/forum/showthread.php?t=315286
     //
24   // The following class defines a hash function for strings
     class stringhasher : public stdext::hash_compare <std::string>
26   {
     public:
28     size_t operator() (const std::string& s) const
       {
30       size_t h = 0;
         std::string::const_iterator p, p_end;
32       for(p = s.begin(), p_end = s.end(); p != p_end; ++p)
         {
34         h = 31 * h + (*p);
         }
36       return h;
       }
38
       bool operator() (const std::string& s1, const std::string& s2) const
40     {
         return s1 < s2;
42     }
     };
44
   public:
46   TokenDB_hashmap(void);
     virtual ~TokenDB_hashmap(void);
48
     virtual bool open( const string &fileName = "" );
50   virtual bool close();
     virtual bool print( ostream &out );
52
     virtual bool addToken( const string &token, int hamCount, int spamCount );
54   virtual bool removeToken( const string &token );
     virtual bool getTokenCounts( const string &token, int &hamCount, int &spamCount );
56   virtual int getDBTokenCount(void) const;
     virtual void mergeDB( TokenDB *db2 );
58
   protected:
60
     struct less_str {
62     bool operator()( const string &x, const string &y ) const
       {
64       return x < y;
       }
66   };

68   hash_map< string, TokenData, stringhasher > m_db;
     virtual void clear(void);
70
   };


   //
2  // TokenDB_hashmap.cpp
   //
4  // class implementation
   //
6  // TokenDB using a hashmap
   //  (microsoft version, since hashmap not officially in STL yet)
8  //
```

```
10 #include "TokenDB_hashmap.hpp"

12 // default constructor
   // intentionally empty
14 TokenDB_hashmap::TokenDB_hashmap(void)
   {}
16
   // destructor
18 // clear the hashmap
   TokenDB_hashmap::~TokenDB_hashmap(void)
20 {
     this->clear();
22 }

24 void TokenDB_hashmap::clear(void)
   {
26   m_db.clear();
   }
28
   // dump the database in format:
30 //  hamCount spamCount tokenString
   bool TokenDB_hashmap::print( ostream &out )
32 {
     hash_map< string, TokenData >::const_iterator iter;
34
     for( iter = m_db.begin(); iter != m_db.end(); ++iter )
36   {
       out << setw(8) << iter->second.m_hamCount
38       << setw(8) << iter->second.m_spamCount
         << "  " << iter->first << endl;
40   }

42   return true;
   }
44
   // open the database,
46 // importing tokens in format:
   //  hamCount spamCount tokenstring
48 //
   bool TokenDB_hashmap::open( const string &fileName )
50 {
     if( fileName == "" )
52     return true;

54   ifstream inFile;
     inFile.open( fileName.c_str() );
56   if( !inFile )
     {
58     return false;
     }
60
     pair< hash_map< string, TokenData >::iterator, bool > mapPair;
62   hash_map< string, TokenData >::iterator mapIter;

64   string token = "";
     unsigned int hamCount, spamCount;
66
     while( inFile >> hamCount >> spamCount >> token )
68   {
       m_db.insert( make_pair( token, TokenData( hamCount, spamCount ) ) );
70   }

72   inFile.close();
     return true;
74 }

76 bool TokenDB_hashmap::close()
   {
```

```
78    this->clear();
80    return true;
  }
82
  // add a token and its counts to the database
84 // check if it exists
  //  if so, then increment the counts
86 //  if not, add it
  bool TokenDB_hashmap::addToken( const string &token, int hamCount, int spamCount )
88 {
    pair< hash_map< string, TokenData >::iterator, bool > mapPair;
90  hash_map< string, TokenData >::iterator mapIter;

92  mapIter = m_db.find( token );
    if( mapIter != m_db.end() ) // it was found
94  {
      mapIter->second.m_hamCount += hamCount;
96    mapIter->second.m_spamCount += spamCount;
    }
98  else // new word
    {
100   m_db.insert( make_pair( token, TokenData( hamCount, spamCount ) ) );
    }
102
    return true;
104 }

106 bool TokenDB_hashmap::removeToken( const string &token )
  {
108   m_db.erase( token );

110   return true;
  }
112
  // given a tokenstring,
114 // return its ham and spam counts
  // returns 0 and 0 if token not found
116 bool TokenDB_hashmap::getTokenCounts( const string &token, int &hamCount, int &
      spamCount )
  {
118   pair< hash_map< string, TokenData >::iterator, bool > mapPair;
    hash_map< string, TokenData >::iterator mapIter;
120
    mapIter = m_db.find( token );
122 if( mapIter != m_db.end() ) // it was found
    {
124   hamCount = mapIter->second.m_hamCount;
      spamCount = mapIter->second.m_spamCount;
126 }
    else // new word
128 {
      hamCount = 0;
130   spamCount = 0;
      return false;
132 }

134   return true;
  }
136
  // how many tokens are in the database?
138 int TokenDB_hashmap::getDBTokenCount(void) const
  {
140   return m_db.size();
  }
142
  // add another DB's tokens to mine
144 void TokenDB_hashmap::mergeDB( TokenDB *db2 )
```

```
{
146   TokenDB_hashmap *mapDB = dynamic_cast<TokenDB_hashmap*>(db2);

148   pair< hash_map< string, TokenData >::iterator, bool > mapPair;
      hash_map< string, TokenData >::iterator mapIter;
150
      for( mapIter = mapDB->m_db.begin();
152     mapIter != mapDB->m_db.end();
        ++mapIter )
154   {
        this->addToken( mapIter->first, mapIter->second.m_hamCount, mapIter->second.
            m_spamCount );
156   }
}


//
// TrainStation.hpp
//
// class interface
//
// this class is in charge of updating the database
//  during initial training and post-classification training
//

#pragma once

#include <algorithm>
#include "Message.hpp"
#include "TokenDB_map.hpp"
#include "TokenDB_hashmap.hpp"
using namespace std;

class TrainStation
{
public:
  TrainStation(void);
  ~TrainStation(void);

  enum TRAIN_MODE
  {
    TEFT,
    TEFT_C,
    TOE,
    NONE
  };

  void train( const Message &msg, Message::MSG_TYPE goldStd, Message::MSG_TYPE
      decision, TokenDB &db );
  void initialTrain( const Message &msg, Message::MSG_TYPE goldStd, TokenDB &db );

  int getCorrectionDelay(void) const;
  void setCorrectionDelay( int delay );

  TRAIN_MODE getTrainMode(void) const;
  void setTrainMode( string mode );

  static const string MESSAGE_COUNTER;

private:
  void trainTEFT( const Message &msg, Message::MSG_TYPE decision, TokenDB &db );
  void trainTEFT_C( const Message &msg, Message::MSG_TYPE goldStd, Message::MSG_TYPE
      decision, TokenDB &db );
  void trainTOE( const Message &msg, Message::MSG_TYPE goldStd, Message::MSG_TYPE
      decision, TokenDB &db );

  void processError( const Message &msg, Message::MSG_TYPE decision, TokenDB &db );

protected:
```

```
    int m_correctionDelay;
    int m_numErrors;
    TRAIN_MODE m_trainMode;
    TokenDB *m_errorTokens;
};

//
// DecisionMatrixFactory.hpp
//
// class interface
//
// this class handles options related to the decision matrix,
// then builds the decision matrix given a message
//

#pragma once

#include "Message.hpp"
#include <vector>
#include <algorithm>
using namespace std;

class DecisionMatrixFactory
{
public:
    DecisionMatrixFactory(void);
    ~DecisionMatrixFactory(void);

    void buildDecisionMatrix( Message &msg, vector<Token*> &decisionMatrix ) const;

    void setMinPrevSightings( int value );
    void setMinMatrixSize( int value );
    void setTokenUsageCount( int value );
    void setHTokenUsageCount( int value );
    void setPTokenUsageCount( int value );
    void setForceInteresting( bool value );
    void setDoubleHamCount( bool value );

protected:
    bool isTokenMature( Token* tok ) const;
    bool isTokenGreat( Token* tok ) const;
    int calcTokenMatrixUsageCount( Token* tok, int currMatrixSize ) const;

protected:
    int m_minPrevSightings;
    int m_minMatrixSize;
    int m_tokenUsageCount;
    int m_hTokenUsageCount;
    int m_pTokenUsageCount;
    bool m_forceInteresting;
    bool m_doubleHamCount;

};

//
// DecisionMatrixFactory.cpp
//
// class implementation
//
// this class handles options related to the decision matrix,
// then builds the decision matrix given a message
//

#include "DecisionMatrixFactory.hpp"

// default constructor
// initializes options
DecisionMatrixFactory::DecisionMatrixFactory(void)
```

```
  : m_minPrevSightings( 5 ),
16   m_minMatrixSize( 15 ),
    m_tokenUsageCount( 1 ),
18   m_hTokenUsageCount( 1 ),
    m_pTokenUsageCount( 1 ),
20   m_forceInteresting( false ),
    m_doubleHamCount( false )
22 {
  }
24
  // destructor
26 // intentionally empty
  DecisionMatrixFactory::~DecisionMatrixFactory(void)
28 {}

30 // buildDecisionMatrix
  // build the decision matrix given a message
32 void DecisionMatrixFactory::buildDecisionMatrix(
    Message &msg,
34   vector<Token*> &decisionMatrix ) const
  {
36   int i=0;
    int j=0;
38   Token *currTok = NULL;
    int usageCount = 0;
40
    // sort the tokens in the message
42   //  see Message::sortMsg() for details
    // important to sort the message,
44   // since calcTokenMatrixUsageCount
    //  just plucks off tokens if there's room in the matrix
46   msg.sortMsg();

48   // consider all tokens in message
    for( i=0;
50     i<msg.getNumTokens();
      ++i )
52   {
      // consider the next token in the message
54     currTok = msg.getToken(i);
      if( currTok == NULL )
56       break;

58     // how many times should the current token be used?
      usageCount = calcTokenMatrixUsageCount( currTok, static_cast<int>(decisionMatrix.
          size()) );
60
      // add the token to the decision matrix
62     //  possibly more than once
      for( j=0;
64       j< usageCount;
        ++j )
66     {
        decisionMatrix.push_back( currTok );
68     }
    }
70 }

72 // determine how many times this token
  //  should be added to the decision matrix
74 int DecisionMatrixFactory::calcTokenMatrixUsageCount( Token* tok, int currMatrixSize
      ) const
  {
76   int usageCount = 0;

78   // has this token been seen before enough
    //  to be considered?
80   // if not, then you can't use the token
```

```
  if( !isTokenMature( tok ) )
    return 0;

  // is the token a special type of token?
  //  if so, then possibly use the appropriate token count
  if( tok->isHeaderToken() )
    usageCount = m_hTokenUsageCount;
  else if( tok->isPhraseToken() )
    usageCount = m_pTokenUsageCount;
  else
    usageCount = m_tokenUsageCount;

  // you can only use the token as many times as it occurs in the message
  usageCount = min( tok->getCount(), m_tokenUsageCount );

  // how many slots remain in the matrix?
  int slotsLeft = m_minMatrixSize - currMatrixSize;

  // are we forcing interesting tokens?
  //  if we're NOT, then we can only add the token
  //  for as many times as slots remain in the matrix.
  if( !m_forceInteresting )
    usageCount = min( slotsLeft, usageCount );
  //  if we are forcing interesting,
  //   check if the token is interesting
  else if( !isTokenGreat( tok ) )
    usageCount = 0;

  return usageCount;
}

// determine token 'maturity'
// used by calcTokenMatrixUsageCount()
// maturity is based on number of times seen before
bool DecisionMatrixFactory::isTokenMature( Token* tok ) const
{
  int prevSightings = 0;


  prevSightings += tok->getHamCount();
  if( m_doubleHamCount ) // graham-like double ham count?
    prevSightings *= 2;

  prevSightings += tok->getSpamCount();

  // have we seen the token enough before?
  if( prevSightings >= m_minPrevSightings )
    return true;

  return false;
}

// determine token 'greatness'
// based on how far the token's score is from 0.5
bool DecisionMatrixFactory::isTokenGreat( Token* tok ) const
{
  return tok->getDistanceFromMean() >= .399999;
}

void DecisionMatrixFactory::setMinPrevSightings( int value )
{
  m_minPrevSightings = value;
}

void DecisionMatrixFactory::setMinMatrixSize( int value )
{
  m_minMatrixSize = value;
}
```

```
150 void DecisionMatrixFactory::setTokenUsageCount( int value )
    {
152   m_tokenUsageCount = value;
    }
154
    void DecisionMatrixFactory::setForceInteresting( bool value )
156 {
      m_forceInteresting = value;
158 }

160 void DecisionMatrixFactory::setDoubleHamCount( bool value )
    {
162   m_doubleHamCount = value;
    }
164
    void DecisionMatrixFactory::setHTokenUsageCount( int value )
166 {
      m_hTokenUsageCount = value;
168 }

170 void DecisionMatrixFactory::setPTokenUsageCount( int value )
    {
172   m_pTokenUsageCount = value;
    }


    //
  2 // TrainStation.cpp
    //
  4 // class implementation
    //
  6 // this class is in charge of updating the database
    //  during initial training and post-classification training
  8 //

 10 #include "TrainStation.hpp"

 12 const string TrainStation::MESSAGE_COUNTER = "__MESSAGE_COUNTER__";

 14 // default constructor
    TrainStation::TrainStation(void)
 16 : m_correctionDelay(1),
      m_numErrors(0),
 18   m_trainMode( TEFT_C )
    {
 20   // currently using a hashmap for the error tokens
      m_errorTokens = new TokenDB_hashmap();
 22 }

 24 // destructor
    // clears errorTokens db
 26 TrainStation::~TrainStation(void)
    {
 28   m_errorTokens->close();
      delete m_errorTokens;
 30 }

 32 int TrainStation::getCorrectionDelay(void) const
    {
 34   return m_correctionDelay;
    }
 36
```

```
void TrainStation::setCorrectionDelay( int delay )
{
  if( delay < 0 )
    return;

  m_correctionDelay = delay;
}

TrainStation::TRAIN_MODE TrainStation::getTrainMode(void) const
{
  return m_trainMode;
}

void TrainStation::setTrainMode( string mode )
{
  transform( mode.begin(), mode.end(), mode.begin(), toupper );

  if( mode == "TEFT" )
    m_trainMode = TEFT;
  else if( mode == "TEFT-C" )
    m_trainMode = TEFT_C;
  else if( mode == "TOE" )
    m_trainMode = TOE;
  else if( mode == "NONE" )
    m_trainMode = NONE;
}

// train the given message
void TrainStation::train( const Message &msg, Message::MSG_TYPE goldStd,
    Message::MSG_TYPE decision, TokenDB &db )
{
  switch( m_trainMode )
  {
  case TEFT:
    trainTEFT( msg, decision, db );
    break;
  case TEFT_C:
    trainTEFT_C( msg, goldStd, decision, db );
    break;
  case TOE:
    trainTOE( msg, goldStd, decision, db );
    break;
  case NONE:
    break;
  }
}

// during initial training
// we just do a simple train-everything
void TrainStation::initialTrain( const Message &msg, Message::MSG_TYPE goldStd
    , TokenDB &db )
{
  trainTEFT( msg, goldStd, db );
}

// train everything - not correctively
void TrainStation::trainTEFT( const Message &msg, Message::MSG_TYPE decision,
    TokenDB &db )
{
```

```
    // train everything,
94  // just add the tokens to the database
    for( int i=0; i<msg.getNumTokens(); ++i )
96  {
      Token *currTok = msg.getToken(i);
98    if( currTok == NULL )
        break;
100
      if( decision == Message::HAM )
102     db.addToken( currTok->getTok(), currTok->getCount(), 0 );
      else if( decision == Message::SPAM )
104     db.addToken( currTok->getTok(), 0, currTok->getCount() );
    }
106
    // increment message counter in DB
108 if( decision == Message::HAM )
      db.addToken( MESSAGE_COUNTER, 1, 0 );
110 else if( decision == Message::SPAM )
      db.addToken( MESSAGE_COUNTER, 0, 1 );
112 }

114 // train everything - correctively
    void TrainStation::trainTEFT_C( const Message &msg, Message::MSG_TYPE goldStd
        , Message::MSG_TYPE decision, TokenDB &db )
116 {
    // train everything.... so go ahead and add the tokens to the main database
118 for( int i=0; i<msg.getNumTokens(); ++i )
    {
120   Token *currTok = msg.getToken(i);
      if( currTok == NULL )
122     break;

124   if( decision == Message::HAM )
        db.addToken( currTok->getTok(), currTok->getCount(), 0 );
126   else if( decision == Message::SPAM )
        db.addToken( currTok->getTok(), 0, currTok->getCount() );
128 }

130 if( decision == Message::HAM )
      db.addToken( MESSAGE_COUNTER, 1, 0 );
132 else if( decision == Message::SPAM )
      db.addToken( MESSAGE_COUNTER, 0, 1 );
134


136 // simulate error correction delay
    //
138 // check for error
    if( goldStd != decision )
140 {
      for( int i=0; i<msg.getNumTokens(); ++i )
142   {
        Token *currTok = msg.getToken(i);
144     if( currTok == NULL )
          break;
146
        if( decision == Message::HAM )
148     {
          m_errorTokens->addToken(
150         currTok->getTok(),
```

```
            -(currTok->getCount()),
152         currTok->getCount()
            );
154       }
          else if( decision == Message::SPAM )
156       {
            m_errorTokens->addToken(
158           currTok->getTok(),
              currTok->getCount(),
160           -(currTok->getCount())
              );
162       }
        }
164
        // update message counter
166     // incorrect ham classification needs to be reversed
        if( decision == Message::HAM )
168     {
          m_errorTokens->addToken( MESSAGE_COUNTER, -1, 1 );
170     }
        // incorrect spam classification needs to be reversed
172     else if( decision == Message::SPAM )
        {
174       m_errorTokens->addToken( MESSAGE_COUNTER, 1, -1 );
        }
176
        ++m_numErrors;
178
        if( m_numErrors == m_correctionDelay )
180     {
          db.mergeDB( m_errorTokens );
182
          m_numErrors = 0;
184       //m_errorTokens.close();
          m_errorTokens->close();
186     }
      }
188 }

190 // train only on error
    void TrainStation::trainTOE( const Message &msg, Message::MSG_TYPE goldStd,
        Message::MSG_TYPE decision, TokenDB &db )
192 {
      // was there an error in judgement?
194   if( goldStd != decision )
      {
196     // yes, there was an error, so process it

198     // put message tokens into error database
        // these tokens were never put into the database as an error before,
200     // so just add them normally
        for( int i=0; i<msg.getNumTokens(); ++i )
202     {
          Token *currTok = msg.getToken(i);
204       if( currTok == NULL )
            break;
206
          if( decision == Message::HAM )
208         m_errorTokens->addToken( currTok->getTok(), 0, currTok->getCount() );
```

```
        else if( decision == Message::SPAM )
          m_errorTokens->addToken( currTok->getTok(), currTok->getCount(), 0 );
      }

      // update the message counter
      if( goldStd == Message::HAM )
        m_errorTokens->addToken( MESSAGE_COUNTER, 1, 0 );
      else if( goldStd == Message::SPAM )
        m_errorTokens->addToken( MESSAGE_COUNTER, 0, 1 );


      // yes, it was an error
      ++m_numErrors;

      // have we seen enough errors to simulate the correction delay?
      if( m_numErrors == m_correctionDelay )
      {
        // add the error tokens to the main database
        db.mergeDB( m_errorTokens );

        m_numErrors = 0;
        m_errorTokens->close();
      }
    }
  }
```

```
  //
  // TestingCenter.hpp
  //
  // class interface
  //
  // automated testing system
  // runs the simulated spamfilter on randomzied index files
  //

#pragma once

#include "SpamFilter.hpp"
#include "IndexMachine.hpp"
#include "TestResults.hpp"
#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/path.hpp>
namespace fs = boost::filesystem;
using namespace std;


class TestingCenter
{
public:
  TestingCenter(void);
  ~TestingCenter(void);

  void runTests(void);
  void setSpamFilter( SpamFilter *sf );
  void setInitialTrainingCount( int count );
  bool setTestSuitePath( string source );
  void setID( string id );
  void setVerbose( int value );

private:
```

```
    void runTest( fs::path indexFile, TestResults &results );

protected:
    SpamFilter *m_sf;
    int m_initialTrainingCount;
    fs::path m_testSuitePath;
    TestResults m_totalTestSuiteResults;
    string m_id;
    int m_verbose;

};

//
// TestingCenter.cpp
//
// class implementation
//
// automated testing system
// runs the simulated spamfilter on randomzied index files
//

#include "TestingCenter.hpp"

// default constructor
TestingCenter::TestingCenter(void)
: m_sf( NULL ),
  m_initialTrainingCount(0),
  m_testSuitePath(""),
  m_id(""),
  m_verbose(0)
{}

// destructor
// intentionally empty
TestingCenter::~TestingCenter(void)
{}

// runTests
// a spamFilter needs to be connected first
// the spam test suite path should also be set
// this methods runs the indexes in the test suite path,
// creating the results files along the way
void TestingCenter::runTests(void)
{
  // we need a spam filter if
  // we plan to do any spam filtering
  if( m_sf == NULL )
  {
    cout << "!!! No spamfilter connected to testing center !!!" << endl;
    return;
  }

  string indexFile;
  int numIndexes = 0;
  ofstream resultsStream;

  // create results directory
  fs::path resultsDirPath = m_testSuitePath / ("Results" + m_id);
  fs::create_directory( resultsDirPath );

  // create main results file
  fs::path resultsFilePath = resultsDirPath / ("Results" + m_id + ".txt");
  resultsStream.open( resultsFilePath.native_file_string().c_str() );
  if( !resultsStream )
  {
    cout << "!! results output file could not be created !!" << endl;
```

```
56      return;
    }
58
    int totalTokenCount = 0;
60
    clock_t stopTime;
62  clock_t singleStartTime;
    clock_t elapsedTime;
64  double elapsedTimeSec;

66  // start overall timer
    clock_t overallStartTime = clock();
68
    fs::directory_iterator end_iter;
70  for( fs::directory_iterator iter( m_testSuitePath );
      iter != end_iter;
72    ++iter )
    {
74    try
      {
76      if( is_directory( *iter ) )
        {}
78      else
        {
80        // check if the filename begins with "Index"

82        indexFile = iter->leaf();
          string::size_type pos;
84        pos = indexFile.find( IndexMachine::getFilePrefix(), 0 );
          if( pos == string.npos )
86          continue;

88        ++numIndexes;

90        // so now we have an index file,
          //  go run a test on that file
92        TestResults currResults;

94        singleStartTime = clock();
          runTest( *iter, currResults );
96        stopTime = clock();
          resultsStream << indexFile << " results"
98          << endl << currResults;

100       elapsedTime = difftime( stopTime, singleStartTime );
          elapsedTimeSec = static_cast<double>(elapsedTime) / CLOCKS_PER_SEC;
102
          int precisionSetting = resultsStream.precision();
104       long flagSettings = resultsStream.flags();
          resultsStream.setf( ios::fixed | ios::showpoint | ios::left );
106       resultsStream.precision( 3 );
          resultsStream << "Time:                    "
108         << elapsedTimeSec / 60 << " min" << endl;
          resultsStream << "DB Token Count:        "
110         << m_sf->getDBTokenCount() << endl << endl ;
          resultsStream.precision( precisionSetting );
112       resultsStream.flags( flagSettings );

114       // upate total results
          m_totalTestSuiteResults = m_totalTestSuiteResults + currResults;
116       totalTokenCount += m_sf->getDBTokenCount();

118       // is verbosity >= 2?
          //  if so, dump the database
120       if( m_verbose >= 2 )
          {
122         fs::path dbPath = resultsDirPath / (indexFile + "_db.txt" );
            ofstream dbStream;
```

```
124           dbStream.open( dbPath.native_file_string().c_str() );
              m_sf->printDB( dbStream );
126           dbStream.clear();
            }
128
            m_sf->resetDB();
130       }
      }
132   catch( const exception &e )
      {
134     cout << "Exception: " << iter->leaf() << " " << e.what() << endl;
      }
136 }

138 // lastly, output the overall results to the results file,
    //  only if there was more then one test run
140 if( numIndexes > 1 )
    {
142   resultsStream << "~~~ COMBINED RESULTS ~~~" << endl;
      resultsStream << m_totalTestSuiteResults;
144
      // stop timer, calculate total time
146   stopTime = clock();
      elapsedTime = difftime( stopTime, overallStartTime );
148   elapsedTimeSec = static_cast<double>(elapsedTime) / CLOCKS_PER_SEC;

150   int precisionSetting = resultsStream.precision();
      long flagSettings = resultsStream.flags();
152   resultsStream.setf( ios::fixed | ios::showpoint | ios::left );
      resultsStream.precision( 3 );
154   resultsStream << "Total Time:           "
        << elapsedTimeSec / 60 << " min" << endl;
156   resultsStream << "Avg DB Token Count:     "
        << totalTokenCount / numIndexes << endl;
158   resultsStream.precision( precisionSetting );
      resultsStream.flags( flagSettings );
160 }

162 resultsStream.close();
  }
164
  // run the test on an individual index file
166 void TestingCenter::runTest( fs::path indexFile, TestResults &currResults )
  {
168   int numMsgProcessed = 0;

170   ifstream indexStream;
    indexStream.open( indexFile.native_file_string().c_str() );
172 if( !indexStream )
    {
174   cout << "index file: " << indexFile.native_file_string()
        << "could not be opened" << endl;
176   return;
    }
178
    // build path to individual test results file
180
    fs::path indexResultsPath = m_testSuitePath
182   / ("Results" + m_id)
      / (indexFile.leaf() + "_results" + m_id + ".txt" );
184
    // open the results file
186 ofstream indexResultsStream;
    indexResultsStream.open( indexResultsPath.native_file_string().c_str() );
188 if( !indexResultsStream )
    {
190   cout << "!! could not create index results file !!" << endl;
      return;
```

```
192   }

194   ofstream matrixResultsStream;
      if( m_verbose >= 1 )
196   {
        fs::path matrixResultsPath = m_testSuitePath
198       / ("Results" + m_id)
          / (indexFile.leaf() + "_matrices" + m_id + ".txt" );
200
        // open the matrix file
202     matrixResultsStream.open( matrixResultsPath.native_file_string().c_str() );
        if( !matrixResultsStream )
204     {
          cout << "!! could not create matrix results file !!" << endl;
206       return;
        }
208   }

210   indexResultsStream.setf( ios::fixed | ios::showpoint | ios::left );
      indexResultsStream.precision( 6 );
212
      string inLine;
214   string goldStdStr;
      string currMessage;
216   string filePath;

218   //Message::MSG_TYPE goldStd = Message::MSG_TYPE::HAM;
      Message::MSG_TYPE goldStd = Message::HAM;
220   Message::MSG_TYPE classification;
      double score;
222
      // loop over index file
224   while( getline( indexStream, inLine ) )
      {
226     // get the goldStd and fileName
        istringstream sStream( inLine );
228     sStream >> goldStdStr >> currMessage;

230     filePath = m_testSuitePath.native_directory_string() + "\\" + currMessage;
        ifstream msgStream;
232
        // try to open the message
234     msgStream.open( filePath.c_str() );
        if( !msgStream )
236     {
          cout << "!! could not open message file: " << filePath << endl;
238       continue;
        }
240
        if( goldStdStr == "HAM" )
242       goldStd = Message::HAM;
        else if( goldStdStr == "SPAM" )
244       goldStd = Message::SPAM;

246     // are we doing initial training?
        if( numMsgProcessed < m_initialTrainingCount )
248     {
          m_sf->initialTrain( msgStream, goldStd );
250     }
        // initial training is over,
252     //  classify the message, then train as normal
        else
254     {
          if( m_verbose >= 1 )
256       {
            matrixResultsStream << currMessage;
258       }
```

```
260         m_sf->classify( msgStream, classification, score, m_verbose,
                matrixResultsStream );
            // reset the message stream back to the beginning of the file
262         msgStream.clear();
            msgStream.seekg(0L);
264         m_sf->train( msgStream, goldStd, classification );

266         // check classification against goldStd,
            //  update current TestResult
268         if( goldStd == classification )
              currResults.incCorrectMsg( classification );
270         else if( goldStd != classification )
              currResults.incWrongMsg( classification );
272
            // print result line to the index results file
274         switch( classification )
            {
276         case Message::HAM:
              indexResultsStream << setw(5) << goldStdStr
278             << setw(5) << "HAM"
                << setw(9) << score
280             << currMessage << endl;
              break;
282         case Message::SPAM:
              indexResultsStream << setw(5) << goldStdStr
284             << setw(5) << "SPAM"
                << setw(9) << score
286             << currMessage << endl;
              break;
288         }
          }
290
          ++numMsgProcessed;
292       msgStream.close();
        }
294
        // close all I/O streams
296     indexStream.close();
        indexResultsStream.close();
298     matrixResultsStream.close();
    }
300
    void TestingCenter::setSpamFilter( SpamFilter *sf )
302 {
      m_sf = sf;
304 }

306 void TestingCenter::setInitialTrainingCount( int count )
    {
308   m_initialTrainingCount = count;
    }
310
    bool TestingCenter::setTestSuitePath( string source )
312 {
      fs::path sourcePath = fs::path( source, fs::native );
314   if( !fs::exists( sourcePath ) )
      {
316     cout << "\n !! Not Found: " << sourcePath.native_file_string() << endl;
        return false;
318   }
      else if( !fs::is_directory( sourcePath ) )
320   {
        cout << "\n !! Test Suite should be a directory of index files !! " << endl;
322     return false;
      }
324   else
      {
326     m_testSuitePath = sourcePath;
```

```
    }
328   return true;
  }
330
  void TestingCenter::setID( string id )
332 {
    m_id = id;
334 }

336 void TestingCenter::setVerbose( int value )
  {
338   if( value >= 0 )
      m_verbose = value;
340 }
```

```
  //
2 // TestResults.hpp
  //
4 // class interface
  //
6 // the current metrics used to gauge
  //  the performance of a particular spamFilter setup
8 //

10 #pragma once

12 #include <iostream>
  #include <fstream>
14 #include "Message.hpp"
  using namespace std;
16
  class TestResults
18 {
  public:
20   TestResults( int hC = 0, int hW = 0, int sC = 0, int sW = 0 );
    ~TestResults(void);
22
    int getNumHam(void) const;
24   int getNumSpam(void) const;
    int getNumMessages(void) const;
26   double getFalsePositiveRate(void) const;
    double getFalseNegativeRate(void) const;
28   double getOverallErrorRate(void) const;
    double getOverallAccuracy(void) const;
30
    void incHamCorrect( int delta = 1 );
32   void incHamWrong( int delta = 1 );
    void incSpamCorrect( int delta = 1 );
34   void incSpamWrong( int delta = 1 );
    void incCorrectMsg( Message::MSG_TYPE type );
36   void incWrongMsg( Message::MSG_TYPE type );

38   TestResults operator+( const TestResults &r2 ) const;
    friend ostream& operator<<( ostream& out, const TestResults &results );
40
  private:
42   int m_hamCorrect;
    int m_hamWrong;
44   int m_spamCorrect;
    int m_spamWrong;
46 };
```

```
  //
2 // TestResults.cpp
  //
4 // class implementations
  //
6 // the current metrics used to gauge
```

```
//  the performance of a particular spamFilter setup
//

#include "TestResults.hpp"

// default constructor
TestResults::TestResults( int hC, int hW, int sC, int sW )
: m_hamCorrect( hC ),
  m_hamWrong( hW ),
  m_spamCorrect( sC ),
  m_spamWrong( sW )
{}

// destructor
// intentionally empty
TestResults::~TestResults(void)
{}

int TestResults::getNumHam(void) const
{
  return m_hamCorrect + m_hamWrong;
}

int TestResults::getNumSpam(void) const
{
  return m_spamCorrect + m_spamWrong;
}

int TestResults::getNumMessages(void) const
{
  return getNumHam() + getNumSpam();
}

// False Positive Rate
//  = numHamWrong / numHam
//
double TestResults::getFalsePositiveRate(void) const
{
  int denom = m_hamCorrect + m_hamWrong;
  if( denom == 0 )
    return 0;

  return m_hamWrong / (double)denom;
}

// False Negative Rate
//  = numSpamWrong / numSpam
//
double TestResults::getFalseNegativeRate(void) const
{
  int denom = m_spamWrong + m_spamCorrect;
  if( denom == 0 )
    return 0;

  return m_spamWrong / (double)denom;
}

// Overall Error Rate
//  = (hamWrong + spamWrong) / numMessages
//
double TestResults::getOverallErrorRate(void) const
{
  if( getNumMessages() == 0 )
    return 0;

  return (m_hamWrong + m_spamWrong ) / (double)getNumMessages();
}

```

```
// Overall Accuracy
//  = 1 - error rate
//
double TestResults::getOverallAccuracy(void) const
{
  return (1.0 - getOverallErrorRate());
}

void TestResults::incHamCorrect( int delta )
{
  m_hamCorrect += delta;
}

void TestResults::incHamWrong( int delta )
{
  m_hamWrong += delta;
}

void TestResults::incSpamCorrect( int delta )
{
  m_spamCorrect += delta;
}

void TestResults::incSpamWrong( int delta )
{
  m_spamWrong += delta;
}

void TestResults::incCorrectMsg( Message::MSG_TYPE type )
{
  switch( type )
  {
  case Message::HAM:
    ++m_hamCorrect;
    break;
  case Message::SPAM:
    ++m_spamCorrect;
    break;
  }
}

void TestResults::incWrongMsg( Message::MSG_TYPE type )
{
  switch( type )
  {
  case Message::HAM:
    ++m_spamWrong;
    break;
  case Message::SPAM:
    ++m_hamWrong;
    break;
  }
}

// operator +
// adds the test results of one run to another
TestResults TestResults::operator+( const TestResults &r2 ) const
{
  return TestResults(
    this->m_hamCorrect + r2.m_hamCorrect,
    this->m_hamWrong + r2.m_hamWrong,
    this->m_spamCorrect + r2.m_spamCorrect,
    this->m_spamWrong + r2.m_spamWrong );
}

ostream& operator<<( ostream &out, const TestResults &results )
{
  int precisionSetting = out.precision();
```

```
    long flagSettings = out.flags();
144
    out.setf( ios::fixed | ios::showpoint | ios::left );
146 out.precision( 6 );

148 out << "Overall Accuracy:      " << results.getOverallAccuracy() << endl;
    out << "False Positive rate:    " << results.getFalsePositiveRate() << endl;
150 out << "False Negative rate:    " << results.getFalseNegativeRate() << endl;
    out << "Total messages:         " << results.getNumMessages() << endl;
152 out << "Ham messages:           " << results.getNumHam() << endl;
    out << "False Positives:        " << results.m_hamWrong << endl;
154 out << "Spam messages:          " << results.getNumSpam() << endl;
    out << "False Negatives:        " << results.m_spamWrong << endl;
156
    out.precision( precisionSetting );
158 out.flags( flagSettings );

160 return out;
}


//
2 // IndexMachine.hpp
//
4 // class interface
//
6 // this class builds random indexes
//  to be used in tests
8 //

10 #pragma once

12 #include <string>
#include <vector>
14 #include <iostream>
#include <fstream>
16 #include <sstream>
#include "Message.hpp"
18 using namespace std;

20 // boost filesystem library
//  used to find all files in a directory
22 #include <boost/filesystem/operations.hpp>
#include <boost/filesystem/path.hpp>
24 namespace fs = boost::filesystem;

26 class IndexMachine
{
28 public:
    IndexMachine(void);
30  ~IndexMachine(void);

32  void addSource( const string &source, Message::MSG_TYPE type );
    void createIndexes(void);
34  void setNumIndexes( int num );

36  static string getFilePrefix(void);

38 private:
    void shuffleSources(void);
40  void dumpSources( ostream &out );

42 protected:
    static string FilePrefix;
44  int m_numIndexes;

46  // store the messages sources as a vector
    //  type of message , path to message
48  vector< pair<Message::MSG_TYPE, fs::path > > m_messages;
```

```
50 };

   //
2  // IndexMachine.cpp
   //
4  // class implementation
   //
6  // this class builds random indexes
   //  to be used in tests
8  //

10 #include "IndexMachine.hpp"

12 // prefix applied to index files
   string IndexMachine::FilePrefix = "index";
14
   // default constructor
16 IndexMachine::IndexMachine(void)
   : m_numIndexes(0)
18 {}

20 // destructor
   IndexMachine::~IndexMachine(void)
22 {
     // clear the messages vector
24   m_messages.clear();
   }
26
   // addSource
28 // opens the given source (should be a directory),
   // then adds the sources to the overall list of sources
30 void IndexMachine::addSource( const string &source, Message::MSG_TYPE type )
   {
32
     // the source might be a file or a folder,
34   // and that source is either ham or spam
     //
36   // add the path of the source (or paths if a folder)
     //  to the message vector
38
     fs::path sourcePath = fs::path( source, fs::native );
40   if( !fs::exists( sourcePath ) )
     {
42     cout << "\n !! Not Found: " << sourcePath.native_file_string() << endl;
       exit(1);
44   }

46   // check if the source is a directory
     if( fs::is_directory( sourcePath ) )
48   {
       fs::directory_iterator end_iter;
50     for( fs::directory_iterator dir_iter( sourcePath );
         dir_iter != end_iter;
52       ++dir_iter )
       {
54       try
         {
56         // for simplicity,
           //  don't allow nested directories
58         if( fs::is_directory( *dir_iter ) )
           {}
60         else
           {
62           m_messages.push_back( make_pair( type, *dir_iter ) );
           }
64       }
         catch( const exception &e )
```

```
66        {
            cout << "Exception: " << dir_iter->leaf() << " " << e.what() << endl;
68        }
        }
70    }
      else // source is just a single file
72    {
        m_messages.push_back( make_pair( type, sourcePath ) );
74    }

76  }

78  // createIndexes
    // we've already added all the desired sources to the messages vector
80  // now we need to actually create the randomized index files
    void IndexMachine::createIndexes(void)
82  {
      string fileName;
84
      cout << endl << " Indexes Created: " << endl;
86
      // for as many indexes as we want...
88    for( int i=1; i<=m_numIndexes; ++i )
      {
90      // randomize the messages
        shuffleSources();
92
        // build index filename
94      fileName = FilePrefix;
        stringstream inStream;
96      inStream << setw(2) << setfill('0') << i;
        fileName += inStream.str();
98      cout << fileName << endl;

100     // open index output stream
        ofstream outFile;
102     outFile.open( fileName.c_str() );

104     // output to index file
        dumpSources( outFile );
106     outFile.close();
      }
108
      cout << endl;
110 }

112 // how many indexes are desired?
    void IndexMachine::setNumIndexes( int num )
114 {
      m_numIndexes = num;
116 }

118 // dumpSources
    // actually output to the index file
120 // the messages have already been randomized
    // the format is:
122 //   MSG_TYPE  relativePathName
    //  where MSG_TYPE is either HAM or SPAM
124 void IndexMachine::dumpSources( ostream &out )
    {
126   for( size_t i=0; i<m_messages.size(); ++i )
      {
128     if( m_messages[i].first == Message::HAM )
          out << "HAM   ";
130     else if( m_messages[i].first == Message::SPAM )
          out << "SPAM  ";
132
        out << m_messages[i].second.native_file_string() << endl;
```

```
134     }
    }
136
    // shuffleSources
138 // simple shuffling function
    void IndexMachine::shuffleSources(void)
140 {
      srand( (unsigned)time(0) );
142
      int RANGE_MIN = 0;
144   int RANGE_MAX = static_cast<int>(m_messages.size());

146   for( size_t i=0; i<m_messages.size(); ++i )
      {
148     int newPos = (rand() % static_cast<int>(m_messages.size()));
        swap( m_messages[i], m_messages[newPos] );
150   }
    }
152
    string IndexMachine::getFilePrefix(void)
154 {
      return FilePrefix;
156 }
```

Vita

Kevin Alan Brown

Candidate for the Degree of

Master of Science

Thesis: A Comparison of Statistical Spam Detection Techniques

Major Field: Computer Science

Biographical

Education: Received Bachelor of Science degree in Computer Science and Mathematics from Southwestern Oklahoma State University in May 2003. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May 2006.

Experience: Employed by the Computer Science Department of Oklahoma State University as a Graduate Teaching Assistant, August 2003 - May 2006.

Name: Kevin Alan Brown Date of Degree: May, 2006

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: A Comparison of Statistical Spam Detection Techniques

Pages in Study: 100 Candidate for the Degree of Master of Science

Major Field: Computer Science

Spam (unsolicited and undesirable email) has become a significant problem for email users. This study investigated the current state-of-the-art in statistical spam filtering. Established methods, inspired by the work of Paul Graham, were examined, and new techniques were introduced and tested. Tests were conducted using two private corpora of email messages and one publicly available corpus.

A base configuration of a spam filter program, similar in technique to a popular production spam filter, was implemented and tested. This configuration achieved high accuracy while maintaining a low false positive rate. One main objective of this paper was to develop a new weighted token probability function. The data contained in header fields are important, and it was believed weighting header data higher than data in the body of the message could improve accuracy. This new weighted token probability function strengthens or weakens header and phrase tokens. Weighting headers applies the weight to any token from a header field, while all body tokens are given unit weight. Weighting phrase tokens keeps the weight of single-word tokens at 1.0, while all remaining tokens of phrase length greater than one are weighted. Tests showed that when tested separately, the header and phrase weights gave mixed results. Also, tests were conducted to show the effects of different initial training set sizes. All three corpora achieved adequate accuracy with small initial training sets, and even performed well with no initial training data, depending on the training method used. Three post-classification training methods and various other techniques were also studied.

Advisor's Approval: J. P. Chandler