2012-03-13

Improving Filtering of Email Phishing Attacks by Using Three-Way Text Classifiers

Alberto Trevino
Brigham Young University - Provo

Improving Filtering of Email Phishing Attacks by Using Three-Way Text Classifiers

Alberto Treviño

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

School of Technology

Brigham Young University

April 2012

ABSTRACT

Improving Filtering of Email Phishing Attacks
by Using Three-Way Text Classifiers

Alberto Treviño
School of Technology, BYU
Master of Science

The Internet has been plagued with endless spam for over 15 years. However, in the last five years spam has morphed from an annoying advertising tool to a social engineering attack vector. Much of today's unwanted email tries to deceive users into replying with passwords, bank account information, or to visit malicious sites which steal login credentials and spread malware. These email-based attacks are known as phishing attacks. Much has been published about these attacks which try to appear real not only to users and subsequently, spam filters. Several sources indicate traditional content filters have a hard time detecting phishing attacks because the emails lack the traditional features and characteristics of spam messages. This thesis tests the hypothesis that by separating the messages into three categories (ham, spam and phish) content filters will yield better filtering performance. Even though experimentation showed three-way classification did not improve performance, several additional premises were tested, including the validity of the claim that phishing emails are too much like legitimate emails and the ability of Naive Bayes classifiers to properly classify emails.

Keywords: Alberto Trevino, email, spam filtering, phish, phishing attacks, support vector machines, maximum entropy, naive bayes, bayesian filters

ACKNOWLEDGMENTS

Special thanks go to my wife and my children. Their support and the desire to provide a better life for them were strong motivators when the work got difficult. I give another special thanks to my wife for putting up with me and doing everything I neglected to do in my family to write this thesis. A special thanks goes to my parents who expected great things from me. Finally, a very respectful and special thanks goes to Dr. Ekstrom for believing in my potential and for fighting for my admission into the program. I also thank my two other committee members for their continual support and their ideas to make this thesis worthy of acceptance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Background

Spam, or bulk unsolicited (often commercial) email has become a large problem on the Internet. Spam consumes network bandwidth, drastically increases load on mail servers, consumes storage space, overwhelms users (the most visible and annoying problem) and provides a delivery vector for security exploits.

The first piece of spam appeared on the ARPANET on May 1, 1978. Shortly after a small amount of spam and chain letters began to appear. But, it was not until 1994 when spamware (software which automated the delivery of high volumes of spam) became available to mark the beginning of the spam we known and endure today. Today around 89% of all email is unwanted and considered spam (MAAGW, 2011). Part of the problem has been how easy spam is to produce and deliver. With today's spamware and botnets, a spammer can easily deliver hundreds of thousands of messages with very little effort.

Another factor contributing to the inundation of spam are the Internet standards (RFCs) which had their origins in the 80s and 90s. Even though they prohibit misinterpreting the origins of email message, they did not provide a way of validating the data until recently. This allowed spammers to hide in a cloak of forgery and deceit which made it hard for Internet provides to find and shut them down.

With the creation of the Spamhaus Project in 1998, the Internet began to fight back against spam. Unfortunately, nearly 18 years later, the fight is far from over.

The first decade of the new millennium has seen the spam battle take on new dimensions. An increasing amount of today's spam has a more sinister purpose than promoting questionable goods and services. Some spam contain and/or link to many types of malware (viruses, backdoors, trojans, etc.) Some spam, pretending to be a reputable source, ask for personal information such as name, address, bank account numbers, credit card information and even usernames and passwords. This type of spam have been labeled as *phishing emails* or *phishing attacks*. The term *phishing* 'originally comes from the analogy that early Internet criminals used email lures to "phish" for passwords and financial data from a sea of Internet users' (Ollmann, n.d.).

Phishing attacks pose a serious security problem that has been difficult to deal with. To be effective, phishing attacks must appear like legitimate messages from a bank, company or acquaintance of the victim. Many sources suggest filtering phishing attacks require different methods than those used for spam since they lack features often found in common spam messages (Bergholz, 2008; Fette 2007; Zdziarski 2006), but there is little consensus on the approach.

1.2 Problem Statement

Security problems caused by phishing attacks are one of the most urgent issues facing System Administrators today. During the first half of 2011, there were over 26,000 unique phishing attacks reported (APWG, 2011). The literature suggests the attackers' success in making the emails look legitimate confuse content filters which then mark the email as legitimate. If phishing emails thwart filters by looking less like spam, then perhaps content filters need to be taught to recognize phish as a separate type of email in order to make them more effective. Most text classifiers (the technology employed in content filters) are capable of classifying text into one of several predefined categories. It should therefore be possible to use multiple categories beyond spam/not spam to train filters to specifically detect what spam, phish and legitimate emails look like in an effort to mitigate the security risk.

1.3 Hypothesis

It is the hypothesis of this thesis that a three-way text classifier can be created and trained to classify incoming email into three separate categories: ham (legitimate), phish and spam, and that through this process the classifier should be able to find better characteristics of what constitutes legitimate email, phishing attacks and spam and therefore improve filtering. In particular, this approach should avoid labeling phishing emails as legitimate simply because they lack the spam features required to label them as spam.

To test this hypothesis I will create labeled corpora of ham, phish and spam. I will also build a feature extractor to extract the elements of each message from these corpora to train and test the classifier. To compare the performance of the three-way classifier, the data will be fed to the text classifier in a traditional two-category configuration where phish and spam corpora will be combined into one large corpus. The filter's performance on the corpora will be compared in the two and three-way configuration with three text classification measures: accuracy, precision and recall in order to determine if using a three-way classifier offered additional performance benefits over the traditional two-way classifier.

1.4 Justification

Even though the literature contains many examples of using text classifiers to identify spam and phishing attacks in serial configurations (first a spam filter followed by a phish filter), there doesn’t seem to be any suggestion that a single three-way classifier has been used to deal with spam and phish simultaneously. This simultaneous approach should provide similar filtering performance to the multistage filters already proposed, but also provide a reduction of computational power required to filter spam and phishing attacks.

1.5 Assumptions

For the purposes of being consistent with much of the literature, and to encourage reproducibility, publicly available corpora will be used for ham, spam and phish emails. These corpora will be assumed to be representative of messages found today in the wild.

The feature extraction process will be generic enough to make sure it can be used with any other corpora.

1.6 Delimitations

Even though the meaning of the term *spam* has evolved to include unsolicited messages to blogs, websites and text messages, this thesis will focus strictly on email messages. No effort will be done to extrapolate the techniques or results in this thesis to other forms of spam.

This thesis will deal exclusively with bulk spam and phish messages. Spear phishing, a highly targeted and focused attack (APWG, 2011) will not be analyzed.

This thesis will not provide statistically meaningful performance comparisons of the many text classifier algorithms. The literature is full of many performance comparisons and this thesis will not attempt to add to it. The criteria for selecting a classifier will be based on finding a classifier with adequate performance, that has the ability to handle multiple categories (or classes), and is freely available for research.

2 REVIEW OF THE LITERATURE

2.1 Introduction to the Literature

Much has been published about spam in the academic, commercial and Internet space. There are several trade and educational conferences dedicated to the subject and many individuals have posted their solutions and frustrations on the Internet. People have even built profitable businesses around that provide spam filtering to other companies and organizations.

The literature converges on three main aspects of spam filtering: standards, heuristics (design and improvement), and machine learning algorithms. Interestingly, very little is published about how these three aspects should be integrated or used in conjunction with each other. Marketing information for several products suggest these aspects are being integrated and combined, but these companies do not provide the details on how it’s being done.

The approach I have taken in my literature review is to analyze the literature to find a research aspect that may not have been published. This requires a full understanding of standards, the history and evolution of spam, and the main solutions that have been proposed and are in use today. This approach provides a solid foundation from which one can begin to understand the problem, the limits of current technologies and identify new areas of research.

2.2 Understanding Email Communications

The email standard which forms the foundation of today's email infrastructure was drafted and published in the form of RFC 821 in August 1982. The standard is titled "Simple Mail Transfer Protocol," abbreviated as SMTP (RFC 821). RFC 821 is not a complete system; it only specifies server-to-server email communication. The entire email system requires at least three other standards in order to work:

- RFC 822: Standard for ARPA Internet Text Messages.
- RFC 974: Mail Routing and the Domain System.
- RFC 918: Post Office Protocol.

RFC 822 specifies how email messages and email addresses are to be constructed. RFC 974 specifies how email servers find each other to deliver email across domains. RFC 821 explains how email servers will talk to one another to deliver email from one server to another. And RFC 974 describes how a user can get email from a mailbox.

2.2.1 Overall Delivery Process

In order to understand email communications, we need to define the components that play a part in it. These components are specified in RFC 5068:

- Mail User Agents (MUA)
- Mail Submission Agents (MSA)
- Mail Transfer Agents (MTA)
- Mail Delivery Agents (MDA)

The overall process can be described as follows. A user authors a message in the Mail User Agent (MUA). Once authored, the MUA sends the message to its configured Mail Submission Agent (MSA). The MSA will query the Mail Exchange (MX) records in DNS for the host where the email will be delivered. Once found, the MSA will contact the responsible Mail Transfer Agent (MTA) and transmit the message via SMTP. Once accepted and received by the MTA, the MTA will pass the message to the Mail Delivery Agent (MDA) where it will be stored in the appropriate mailbox(es) of the recipient(s). From there, the recipient(s) will obtain the message via POP or other means through his/her MUA where the message will be read (RFC 5068). The Kavi Help Center provides a diagram of this process, shown in Figure 2-1.

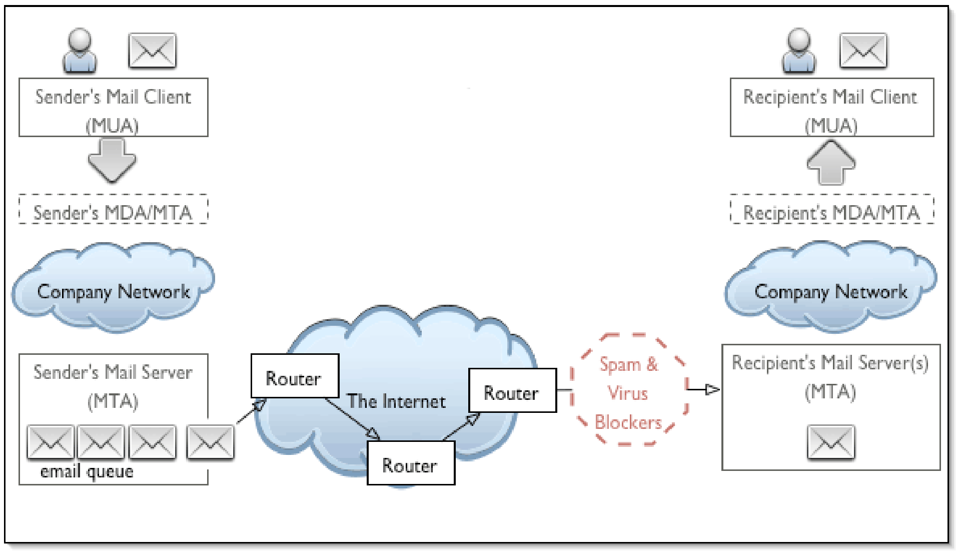

Figure 2-1: Email Delivery Process

One crucial difference of today's email delivery practices from those of the early Internet and ARPANET, is relaying. During the early days of the ARPANET automatic, routed connections were not possible between many of the available nodes. It was therefore a common practice to relay email through several MTAs from one server to the next. Thus, an email message may have passed through several SMTP servers before reaching its final destination. This crude but necessary practice could have made the tracing and debugging of email delivery problems very difficult. However, the first RFCs (821 and 822 in particular) were designed to accommodate logging and tracing of the many paths the email may have taken. In Section 4.1.1 RFC 821 states:

> When the receiver-SMTP accepts a message either for relaying or for final delivery it inserts at the beginning of the mail data a time stamp line. The time stamp line indicates the identity of the host that sent the message, and the identity of the host that received the message (and is inserting this time stamp), and the date and time the message was received. Relayed messages will have multiple time stamp lines.
>
> When the receiver-SMTP makes the "final delivery" of a message it inserts at the beginning of the mail data a return path line. The return path line preserves the information in the <reverse-path> from the MAIL command. Here, final delivery means the message leaves the SMTP world. Normally, this would mean it has been delivered to the destination user, but in some cases it may be further processed and transmitted by another mail system (RFC 821).

It is important to note none of the email standards provide mechanisms for *verifying* the route taken by an email message. The use of the “Received” and “Return-Path” headers was designed to provide this information as a debugging tool. It was never meant to a forensic tool.

2.2.2 The SMTP Session

The Simple Mail Transfer Protocol is the heart of email delivery. It is often used by the MSA to start message delivery, and it is used by all public MTAs to deliver email throughout the Internet. SMTP is the glue that binds all email delivery.

The SMTP session is a lot like a telephone conversation. A typical SMTP session starts with the two sides exchanging greetings. Then, the SMTP client (the system sending the email message, usually the MSA) specifies who is sending the email, followed by a list of recipients. Once the source and destinations have been exchanged, the actual mail message is transferred at which point the receiving end (the SMTP server) either acknowledges the reception of the email or rejects it, either temporarily or permanently. The SMTP session is, to a degree, human readable. Figure 2-2 shows a modern, sample SMTP session as specified in RFC 5321.

```
This SMTP example shows mail sent by Smith at host bar.com, and to
Jones, Green, and Brown at host foo.com.  Here we assume that host
bar.com contacts host foo.com directly.  The mail is accepted for
Jones and Brown.  Green does not have a mailbox at host foo.com.

      S: 220 foo.com Simple Mail Transfer Service Ready
      C: EHLO bar.com
      S: 250-foo.com greets bar.com
      S: 250-8BITMIME
      S: 250-SIZE
      S: 250-DSN
      S: 250 HELP
      C: MAIL FROM:<Smith@bar.com>
      S: 250 OK
      C: RCPT TO:<Jones@foo.com>
      S: 250 OK
      C: RCPT TO:<Green@foo.com>
      S: 550 No such user here
      C: RCPT TO:<Brown@foo.com>
      S: 250 OK
      C: DATA
      S: 354 Start mail input; end with <CRLF>.<CRLF>
      C: Blah blah blah...
      C: ...etc. etc. etc.
      C: .
      S: 250 OK
      C: QUIT
      S: 221 foo.com Service closing transmission channel
```

Figure 2-2: Example SMTP Transmission

RFC 821 and its subsequent revisions (2821 and 5321) specify that a server should be rather forgiving, following the Internet tradition of making communications as usable and reliable as possible. For example, commands are not case-sensitive and any command errors should be properly handled. Even though some rules are fairly strict (for example, the command verbs) the predicates of such verbs are relaxed (RFC 5321). Here is a summarized list of the minimum commands that should be supported by an SMTP server, and how they are to be transmitted by the client.

Table 2-1: Minimum Set of Commands Required to Implement an SMTP Server and Their Descriptions

Command	**Description**
EHLO	EHLO (extended hello or HELO for backward compatibility) must be the first command sent by a client with the Fully Qualified Domain Name (FQDN) of the client or, its IP address.
VRFY	Checks (verifies) if an email address is valid for a particular domain (RFC 821). RFC 5321 suggests this feature might be turned off on most systems since it is believed to be abused by spammers to harvest email addresses. But the same RFC emphasizes that it should be enabled and properly verify addresses.
MAIL FROM	Specifies the email address responsible for sending the email. This gets recorded as the Return-Path in the headers by the receiving SMTP server.
RCPT TO	Specifies the recipients for the message. Multiple addresses can be given and they can be individually accepted or rejected by the SMTP server.
DATA	Sends the headers and body of the email. After the data has been set, another session can be initiated by using the RSET command or by sending another MAIL FROM command.
RSET	Resets the connection back to the state established after initial EHLO/HELO.
NOOP	Sends a command that does nothing, usually to keep the connection alive.
QUIT	Used to end the session and close the connection.

There are a few things to notice about the standard, mostly from the things it does not contain. SMTP does not require any type of authentication. Extensions for it exist, and RFC 5068 recommends the use of authentication for the submission of email from the MUA to the MSA preferably over TCP port 587 (RFC 5068). Another purposeful omission was the verification of the email itself. The SMTP server expects the message to be properly formatted as stated in RFC 5322, but it is not required to verify it. Verifications on the fully qualified domain name (FQDN) of the HELO/EHLO command, or verifications on the existence of the MAIL FROM address are also missing.

One thing the standard did address properly is relaying. Any SMTP client can *ask* any SMTP server to send a message to any recipient outside the domains it handles. The server has the prerogative to accept and relay the message, or reject the recipient. There are two reasons why an SMTP server would reject a recipient: the recipient does not exist on that server, or the server does not handle nor relay email to that particular domain. Spammers often try to relay spam in order to avoid being blacklisted (see Section 2.5.2) However, the SMTP standard does not require SMTP servers to support relaying. It is highly recommended that SMTP servers disallow relaying to avoid being abused by spammers (RFC 2505).

Even though relaying was once a necessity, today it is no longer necessary. In RFC 5321 we read:

> In general, the availability of Mail eXchanger records in the domain name system makes the use of explicit source routes in the Internet mail system unnecessary. Many historical problems with the interpretation of explicit source routes have made their use undesirable. SMTP clients SHOULD NOT generate explicit source routes except under unusual circumstances. SMTP servers MAY decline to act as mail relays or to accept addresses that specify source routes (RFC 5321).

One other aspect of SMTP which should not be ignored are the three possible responses which a server can provide to an incoming message: accepted (2yz response code), temporary failure (4xx response code) and permanent failure (5yz response code). Note there is no "message received, please wait while we run your message through our spam filters" option. That means MTAs must make a relatively quick decision as to whether to accept or reject a message. RFC 2821 includes a section minimum SMTP command timeouts. These timeouts are to be used per command and not for the entire transmission. Section 4.5.3.2 of RFC 2821 states the SMTP client should wait a minimum of 10 minutes after the message has been sent for a response from the SMTP server, before considering the session lost (RFC 2821). This provides SMTP servers with at least 10 minutes to process the email and decide whether it is spam or not.

2.2.3 Sending Spam

Sending spam is not much different than sending legitimate email. "The very characteristics that make email such a convenient communications medium -- its near ubiquity, rapid delivery, low cost, and support for exchanges without prior arrangement -- have made it a fertile ground for the distribution of unwanted or malicious content" (RFC 5068). From the SMTP perspective, there is no difference between legitimate email and spam. SMTP, as designed, does not require or allow for the transmission of intent, which is the only difference between legitimate email and spam. This is why spam is so difficult to accurately detect. However, the many spam detection techniques explained later on are intended to do just that.

One aspect of the SMTP process in particular has proved to be detrimental to spammers: its reliability. The SMTP standard specifies that if an email message was not delivered, a notice must be sent back to the sender notifying him/her of the error (RFC 5321). If a spammer is not carefully, he/she could quickly be inundated with error messages (bounce notices). To avoid the flood of bounced messages, spammers learned early on they could provide false return information on the email's envelope and redirect these bounces to another recipient or even to non-existent email addresses. Additionally, spammers hide their tracks by providing erroneous or invalid HELO and MAIL FROM information. An SMTP server implemented as specified by the SMTP standard does not care if it is lied to. In fact, it is designed to be forgiving of mistakes and to accept anything that may be good enough to be delivered. “Utility and predictability of the Internet mail system requires that messages that can be delivered should be delivered, regardless of any syntax or other faults associated with those messages and regardless of their content (RFC 5321). Spammers, of course, have exploited the good will of the Internet in this regard.

2.3 History of Spam

The first piece of spam appeared on May 1, 1978 as an advertisement for a product demonstration by Digital Equipment Corporation over the ARPANET (Zdziarski 2005). The reaction to this first piece of spam (although the term *spam* had not be coined yet) was quite strong by the ARPANET community and the ARPANET’s managers, the U.S. Department of Defense. In one reply Major Raymond Czahor stated the DEC message “was a flagrant violation of the use of ARPANET as the network is to be used for official U.S. government business only” (Templeton, n.d.).

During the 1980s bulk emails such as chain letters and religious messages appeared from time to time in slowly increasing volume. In 1994 a husband and wife attorney team “decided to hire a programmer who could write software to post an advertisement to every single newsgroup in existence” (Zdziarski, 2005). The email, referred in the literature as the Canter & Siegel spam, promoted the husband and wife’s immigration legal services. Their idea of automating the posting of emails was revolutionary since before this time spam was sent by entering individual email addresses by hand to a new message. In 1994 and 1995 spamware, or automated spam software, hit the market and several large spamming companies emerged (Zdziarski, 2005).

Around 2003 a shift began on how spam was delivered, from relatively small distribution points to fully distributed methods through botnets (Seabrook, 2008). (Botnets are defined by Wikipedia as a collection of compromised computers on the Internet.) By 2008 it was believed that 85% of all spam originated from only six different botnets (M86 Security, 2008). With so few botnets responsible for such a large amount of spam, one may quickly think it would be an easy feat to end spam. That is not the case, however. Botnets are extremely resilient systems. They can be considered the Internet equivalent of organized crime and, as soon as one system is shut down, another quickly takes its place (Leyden, 2012).

2.4 Types of Spam

Although the term spam is used loosely as a general term to describe unwanted email, it is important to distinguish the many different variations of spam.

2.4.1 Unsolicited Commercial Email

Unsolicited Commercial Email (UCE) was the most common type of spam of the mid to late 1990's and early 2000's. Its primary purpose is to promote goods and services. They often promote a variety of services including adult content, health, IT, personal finance, education and training, politics and even anti-spam solutions (Securelist, n.d.).

The rise of UCE during the 1990's also led to a rise of questionable, underground companies. One common example are emails promoting sales of male enlargement pills through rogue online pharmacies. One other example common in the mid 2000's was stock promotion which would send “positive ‘touts’ for a low-volume security in order to manipulate its price and thereby profit on an existing position in the stock” (Kanich, 2009).

2.4.2 Chain Letters

Chain letters became very popular in the early days of the Internet (1980’s to early 1990’s) (Zdziarski 2005). They basically asked users to proliferate the email to a number of other people in order to obtain good luck. Although not commercial, these types of emails are considered unwanted by most.

2.4.3 Scams

There are many scams sent by email. Some of the most popular scams are the Nigerian Letter scams which found new life in the digital age. The FBI describes these scams on their website:

> Nigerian letter frauds combine the threat of impersonation fraud with a variation of an advance fee scheme in which a letter mailed from Nigeria offers the recipient the "opportunity" to share in a percentage of millions of dollars that the author—a self-proclaimed government official—is trying to transfer illegally out of Nigeria. The recipient is encouraged to send information to the author, such as blank letterhead stationery, bank name and account numbers, and other identifying information using a fax number provided in the letter. Some of these letters have also been received via e-mail through the Internet. The scheme relies on convincing a willing victim, who has demonstrated a "propensity for larceny" by responding to the invitation, to send money to the author of the letter in Nigeria in several installments of increasing amounts for a variety of reasons.
>
> Payment of taxes, bribes to government officials, and legal fees are often described in great detail with the promise that all expenses will be reimbursed as soon as the funds are spirited out of Nigeria. In actuality, the millions of dollars do not exist, and the victim eventually ends up with nothing but loss. Once the victim stops sending money, the perpetrators have been known to use the personal information and checks that they received to impersonate the victim, draining bank accounts and credit card balances.

2.4.4 Malware Proliferation

Much of today's spam is intended to spread malware. These messages purport to have information regarding a package delivery, offer software licenses, or even have information about tax payments or failed bank transfers (Websense, n.d.). These messages often contain infected attachments or links to infected websites to aid in the spread of various types of malware.

2.4.5 Phishing Attacks

Phishing attacks try to extract private information from users by creating legitimate-looking emails from common organizations which directly ask for username, passwords, account numbers, etc. or link to legitimate looking websites which collect that information from users. Some pretend to be from PayPal, eBay, and wide variety of banks. These messages are bulk, and cast a wide net and often are sent to people who may not do business with the company they pretend to represent. In contrast, a targeted form of phishing attack is called *spear phishing* which, as described by the FBI on their website,

> [targets] select groups of people with something in common—they work at the same company, bank at the same financial institution, attend the same college, order merchandise from the same website, etc. The e-mails are ostensibly sent from organizations or individuals the potential victims would normally get e-mails from, making them even more deceptive. . .
>
> First, criminals need some inside information on their targets to convince them the e-mails are legitimate. They often obtain it by hacking into an organization's computer network (which is what happened in the above case) or sometimes by combing through other websites, blogs, and social networking sites.
>
> Then, they send e-mails that look like the real thing to targeted victims, offering all sorts of urgent and legitimate-sounding explanations as to why they need your personal data.
>
> Finally, the victims are asked to click on a link inside the e-mail that takes them to a phony but realistic-looking website, where they are asked to provide passwords, account numbers, user IDs, access codes, PINs, etc.

2.5 Proposed Solutions

There have been many solutions proposed to solving the spam problem. This section attempts to summarize some of the known approaches explaining how they work, their benefits and their shortcomings.

2.5.1 Keywords, Phrases and Regular Expressions

One of the first solutions to combat spam was the use of keywords and phrases (Zdziarski, 2005). This solution is simple. A list of keywords, phrases, or even regular expressions commonly found in spam is run through the filter. If the phrase is present in the message, the message is labeled as spam.

This method is not considered very effective. Even though it can be very efficient at blocking certain emails, it can also have a high false positive rate (legitimate emails marked as spam). This method also requires large administrative resources since each unique spam must be individually dissected and programmed. False positives also have to be individually handled. In all this manual tuning, it is very easy to have unintended consequences which can be very difficult to correct (Zdziarski, 2005). Because of its high administrative cost and low performance, this type of filtering is strongly discouraged.

2.5.2 Blacklists

Blacklists are a simple but effective approach. The idea is to make list of IP addresses which are known to send spam and block them outright (Jacob, 2003). Blacklists can either be manually created by each organization or obtained from blacklist services. In 1996 Spamhous became the first non-profit blacklist provider (Zdziarski, 2005).

Blacklists are not perfect, however. For any given IP address it is an all-or-nothing approach. There is also the inconvenience of being wrongfully listed in a blacklist. This can happen due to abusive behavior by someone who has used the IP address previously, or whom you may share the IP address with in some hosting scenarios, or simply because the blacklist administrator felt like it (Jacob, 2003).

The introduction of IPv6 threatens to destabilize blacklists. Since blacklist are stored mostly by IP address, the new addressing scheme will force blacklist administrators to recreate a lot of their work (Leyden, 2011).

2.5.3 Whitelists

Whitelists, the reverse of blacklists, provide a list of IP addresses that are allowed to send email. If a particular IP address is not on the list, it will be blocked and not allowed to send email. As with blacklists, whitelists are hard to maintain, are not exhaustive by nature, and tend to block much more than just spam (Zdziarski, 2005). They can have a legitimate use in closed systems that only talk to other trusted systems.

2.5.4 Heuristics

Heuristics is a general term used to describe rules or checks performed on email. An example would be checking for a valid, fully qualified domain name (FQDN) in the HELO/EHLO command. Heuristics are a mixed bag. They can perform anywhere from really well to very poorly. One way to improve performance is to combine multiple heuristics with a point-based system so that no one heuristic is usually entirely responsible for flagging an email as spam. The decision is deferred until all heuristics have been considered, adding up and subtracting all the necessary points. If the total points is higher than some configurable threshold, the email is flagged as spam and rejected. (Zdziarski, 2005).

SpamAssassin, perhaps the most common spam fighting tool in use, is heuristic based. In version 3.3.x (current stable branch at the time of writing) SpamAssassin comes with 356 different heuristics. They check a variety of things from header structure, keywords and phrases, remote client identification, etc. (SpamAssassin Documentation, n.d.).

Proper heuristics offer acceptable performance although they carry a very high administrative cost. Heuristics are nearly always hand-written, and even though their point systems can be tuned with algorithms, often times they are manually adjusted (Youngman, 2004).

2.5.5 Machine Classifiers

Machine classifiers are often used in Natural Language Processing. They are capable of classifying text into predefined categories. Machine classifiers are usually built on top of machine learning algorithms to create probabilistic models for each category. When a classifier classifies text similar to the one given in training, it returns the most probable category for the text (Manning, 2004). There are many different types of machine classifiers, such as Naive Bayes, CRM114 (Zdziarski, 2005), Maximum Entropy (Manning, 2004), Winnow (Chhabra, 2005), Decision Trees, Random Forests, Neural Networks (Caruana, 2006) and Support Vector machines (SVMs) (Joachims, 2002). Each method is usually built to solve a particular problem, so their performance can vary greatly depending on the problem they are applied to. (See Carauna, 2006 for explanations and comparisons between many of these classifiers.)

Classifiers, when used as spam filters, are usually configured with two categories or classes: spam and not spam. SpamBayes offers an exception, adding the ability to return an *unsure* classification for messages which are difficult to predict (Spambayes, n.d.).

Overall, text classifiers can be classified into two categories, depending on the statistical model they use. These are generative (such as Naive Bayes and CRM114) or non-generative (such as Maximum Entropy, Winnow, and SVM). Generative models get their name from the ability to generate new, representative documents for each category in their models. In general, they keep separate statistical models for each defined category. Non-generative models (also known as *conditional models* and *discriminatively trained models*) are not able to generate any documents. In contrast to generative models, non-generative models contain a single statistical model which, during the training phase, systematically find the differences in the documents that help determine the category they belong to. Non-generative models are able to calculate the *posterior probability* (the probability a document belongs to a particular category given its features) directly, while generative models must do it from the *joint (generative) distribution* using Bayes' Law (Ringger, 2012).

One aspect affecting the performance of classifiers is feature extraction. The literature seems to favor single word features which represent email messages by their individual words. Word features work fairly well, but can be driven into feature deprivation, a condition where features have never been seen in training and are not accounted for in the model (Kanaris, 2006).

Another approach which reduces the possibility of feature deprivation is using character-based features. Even though it may seem counter intuitive, character-based spam filtering can still be quite effective. One benefit of character-based features is that the feature space is theoretically smaller, meaning there is a higher probability of accounting for most features during training (Kanaris, 2006).

2.5.6 Challenge and Response Systems

An idea suggested and implemented in the late 1990s was the challenge/response model. Basically, whenever a new sender would try to deliver email to a new mailbox, the system would place the delivery of the message on hold and automatically reply to the sender with a confirmation email. If the original sender wanted the email delivered, the sender would have to reply to the confirmation email using the same email address of the original message. On successful confirmation, the original email would be delivered to the recipient. In case several emails would arrive from the same sender only one confirmation email would go out (if designed properly) and would all be released upon confirmation. The reasoning behind this approach was that real senders would receive the confirmation email and would reply, while spammers would not be interested in replying to every single confirmation from all the spam they had sent. In a way, proper delivery required cooperation from the sender and would shift some of the burden back on spammers (Templeton, n.d.)

This idea worked a little too well: the user would see little flowing into his/her inbox, both in terms of email and spam. Most legitimate senders would reply , and some spammers as well. But, just like this approach put an extra burden on spammers, it also added to the burden of regular users as well. Some people refused to reply to these automated messages (SpamCop FAQ, n.d.).

In terms of solving all the problems with spam, this approach relieved inboxes (often relieving them too much) but also increased bandwidth, since many messages would require a confirmation email. Storage requirements didn't really reduce since much of the spam was stored awaiting confirmation. Finally, there was the issue of forgery. Since spammers often forge email addresses, confirmation emails were either sent to unsuspecting users or were bounced by the innocent domains (SpamCop FAQ, n.d.). Thus, this approach has been mostly abandoned.

2.5.7 Greylisting

Greylisting can be considered a variation on white lists, black lists and challenge/response systems, except that it all happens at the machine level, not the user level. The idea behind it is that any new sender will be temporarily blocked until the sender's identity can be confirmed. Instead of sending a confirmation email to the sender, the receiving SMTP server temporarily denies the message with a 4yz response after the message has been received. This situation forces the sender's server to retry sending the same message some time later. If the resent messages matches the first attempt (which it should) then the user is added to the whitelist and the original message is delivered. If, however, the message is not resent then the message is never delivered. This method works because most spam is not resent; resending spam would go against the economics of bulk deliveries (Yamai, 2008).

This method has proved very effective. However, it is not perfect. First, additional bandwidth is created by forcing the email server to resend a message. This problem is exacerbated with large attachments. Additionally, extra care needs to be taken when deciding what constitutes the same message. Large organizations with multiple outgoing mail servers cannot guarantee the email will be resent by the same server. Thus, precautions need to be made to ensure the message can be delivered from multiple servers. Finally, greylisting introduces noticeable delays in the delivery queue. Standards suggest waiting at least 30 minutes before resending the message, thus adding 30 minutes to the deliver of email (RFC 2821).

In an effort to improve on the delay aspect of greylisting, Yamai suggests that perhaps closing the connection without warning (instead of providing a proper 4xx response) may be a better approach. Based on his presentation, a dropped connection triggers a near immediate retry from the server, removing the time delay. Additionally, it is usually retried from the same server, removing the problem of retransmissions from another host (Yamai, 2008).

2.5.8 Tarpitting

Tarpitting is another technique aimed at hurting the cost margins of spam. This technique was developed in the mid 2000's and was aimed at the many penny stock scams. The technique works by purposely slowing down communications between the client and the server, thereby slowing the spammer. With the incredibly low reply rates (estimated to 0.003% by Kanich, 2008) a spammer must send millions of messages in order to make a profit. By tarpitting, the spammer is unable to send the large amounts of spam required to make a profit in a fast enough time frame (Eggendorfer, 2007).

In order to understand this technique better, it is important to understand the delivery times involved and the suggested latencies clients should expect. A regular email message (one without large attachments) between moderately busy servers can be delivered within a few seconds. However, when combined, the minimum timeout periods suggested in RFC 2821 provide up to 30 minutes in wait times for a single message. If a simple message can be delivered in less than 3 seconds but SMTP clients should be designed to wait at least 1,800 seconds, that means one could potentially slow down the deliver of spam by a factor of 600. Of course, legitimate email can also be slowed down to those levels. In practice 30 minute delays are not typical. Even delays of about five minutes are enough to deter impatient spammers while still allowing legitimate mail to come through at a more regular pace.

There are two drawbacks with this solution. First, the delay imposed on all email. The second, the extra load on mail servers since they will have to cope with a much larger number of simultaneous connections (Eggendorfer, 2007).

2.5.9 Sender Policy Framework (SPF)

Sender Policy Framework (or SPF) is one of the first standard-based approaches to the spam problem. SPF began as a proper solution for authenticating outgoing mail servers for a particular domain. SPF had its starts with an idea by Hadmut Danisch of publishing Reverse MX (RMX) records through DNS. That way, an SMTP server could get a list of valid outgoing mail servers in a way similar to how they get a list of incoming mail servers. The basic idea of RMX was to reduce forgery by having a way to authenticate whether a particular client attempting to send email was a valid outgoing server for that domain.

SPF took the basics of MX and improved on them by allowing the publishing of policies regarding outgoing email for a particular domain, authorizing or blocking IP addresses and subnets. SPF also allows for simple configurations for domains where the outgoing servers are the same as the incoming mail servers by simply authorizing all servers listed in the MX record (RFC 4408). The different SPF policies (or validation outcomes) allow for great flexibility in how to deal with apparent rouge email. The SPF policies specified in RFC 4408 are:

- PASS: The specified IP or host is allowed to send email for the domain
- NEUTRAL: The sender is neither authorized to or prohibited from sending email for the domain.
- SOFT FAIL: The use of this host is discouraged sending email for the domain.
- HARD FAIL: The host is prohibited from sending email for the domain.

SPF has had some success. Some people don't think it has done anything at all. Transient media reports early on seem to suggest more spammers were using SPF than legitimate domains. This seems to be a continuous problem as J. Goodman from Microsoft reported in 2007 40% of the servers that sent email to Hotmail had SenderID records (Goodman, 2007). But SPF seems to get its bad reputation from a misunderstanding of what the technology actually does. When it was first released, it was touted as the solution that would end spam . That obviously did not happen. For that reason, many dismissed it. SPF is no miracle cure, but it does exactly what it was intended to do: authenticate sender MTAs for a domain. Of course, spammers can authenticate as well, and the biggest complaint has its source at unconditionally trusting SPF records, or not doing additional checks with neutral policies which should be considered as having no SPF record at all (RFC 4408).

2.5.10 Sender ID

Sender ID is a variation of SPF developed by Microsoft which can authenticate the MAIL FROM field along with what it calls the Purported Responsible Address (PRA) obtained from the headers. Sender ID works in a very similar ways to SPF by using DNS records to publish its information. (RFC 4406).

SenderID has not been very popular outside of the Microsoft partners since several of its technologies are patent-incumbered and cannot be implemented in free software projects (Apache Foundation, 2004). For those projects SPF is preferred.

2.5.11 Relay Header Detection

In cases where SPF or SenderID do not have any records, there are other DNS queries which can be performed to ascertain the validity of the EHLO greeting and MAIL FROM commands. Relay Header Detection specifies the order and type of DNS queries which can be performed to ensure the message comes from a reputable source. In testing, this technique was able to filter email with over 99% accuracy with very low false positives (Treviño, 2007b).

2.5.12 DomainKeys Identified Email

DomainKeys Identified Email (DKIM) is a digital signature system for email created by Yahoo!. It serves as a way of ensuring that a valid server for a domain has sent a particular message. It works by adding the DKIM-Signature email header containing the domain information along with a digital signature to the message. The receiving MTA can then obtain a copy of the public key and verify the digital signature (RFC 4871).

Just like SPF and Sender ID, DKIM suffers from adoption problems. Additionally, having a signed email does not mean you won't have spam. Spammers can sign messages too. And, ironically, Yahoo! is responsible for spam itself when spammers create and use Yahoo! accounts to distribute spam. However, its weaknesses should not deter its use .

2.5.13 DMARC

On January 30, 2012 Google, Facebook, Microsoft, PayPal and several other Internet giants unveiled a new plan to combat phishing attacks. Their approach called *Domain-based Message Authentication, Reporting & Conformance* (DMARC) hopes to improve on existing SPF/SenderID and DKIM standards. From the DMARC website:

> DMARC standardizes how email receivers perform email authentication using the well-known SPF and DKIM mechanisms. This means that senders will experience consistent authentication results for their messages at AOL, Gmail, Hotmail, Yahoo! and any other email receiver implementing DMARC. We hope this will encourage senders to more broadly authenticate their outbound email which can make email a more reliable way to communicate.

The number of industry heavyweights which have come together to form this new approach is a testament to the extent and difficulty of dealing with the phishing problem. With time we will be able to see how much this new approach will help reduce the number of phishing attacks.

2.5.14 Legislation

Several countries including the United States and the European Union have enacted laws regulating the mass distribution of unsolicited, commercial email and providing punishments for those who break them. The U.S. law (known as the CAN SPAM Act) was signed into law in December 2003. Since then, it has provided means for Internet Service Providers (ISPs) and law enforcement to charge, prosecute and incarcerate spammers. Since their inception, several prominent spammers have been successfully convicted. One well-known example was the conviction of Alan Raisky, known for running many stock fraud scams (Leyden, 2008).

The laws in the United States don’t ban all bulk or commercial emails. However, it states that bulk commercial email must provide the following:

1. Legitimate information about the sender in the headers.
2. Mention the email is an advertisement.
3. Instructions on how to be removed from the sender’s list.
4. Human-readable information of the origins of the message. (FTC).

Legislation by itself has done little to stop spam, however. It would appear most spammers are not concerned about breaking any laws. Before federal legislation was passed in the United States, several of its states had created their own laws and often conflicted with each other. These conflicts made it hard for spammers to simultaneously comply with multiple laws in one message, giving them an extra reason to ignore such laws.

2.5.15 SMTP Proxies

SMTP proxies act as intermediary SMTP servers between Internet SMTP servers and an organization’s incoming SMTP servers. By being in the middle they attempt to catch spam before it gets to the SMTP server that will deliver email to users. The most interesting implementation is a product called ASSP (or @SSP). It stands for Anti-Spam SMTP Proxy. It is written in Perl and sits and listens to incoming SMTP connections and relays commands to a standard SMTP server listening on another address or port. ASSP then applies most of the techniques described so far to try to filter spam by ensuring it never reaches the actual SMTP server (Hanna, n.d.).

ASSP, unfortunately lacks good documentation. It has too many options without any explanation of how they worked, or how they should be (or shouldn’t be) combined. Overall, this solution followed the recommendation made in RFC 2505 to implement spam filtering at the SMTP level. Additionally, by implementing a command proxy model, the developers were able to ensure quick deployment and compatibility with all SMTP servers without having to write delivery modules.

2.6 Current State of the Commercial Industry

The IT industry has created some formidable anti-spam solutions. Some of these solutions have been so effective one editor of Communications of the ACM wondered if the spam war had been won (Dacier, 2009). That's the real advantage of an commercial solution: if you have the budget, you can hire a company to take care of the problem for you. This type of solution is very appealing for many organizations since it will virtually eliminate spam, and won't require investments in infrastructure, personnel and training. However, these solutions are not a panacea. First, there are the contract fees for their services. Additionally, the organization will lose control over their email and filtering. Granted, these services give the users some control, but if what you want to do is outside of the company's model, you may be out of luck. Then there is fear. If an organization deems its email communications as too valuable to hand over to an external company, they may have to continue to do their spam filtering in-house.

There is also the arms race between companies. Most of these companies will not disclose their methods. This secrecy greatly hurts the academic world since it is very difficult to pursue academic research without stepping on some toes or someone's cash cow.

Here is a quick review of some of the most popular commercial solutions.

2.6.1 Barracuda Networks

Barracuda Networks provides a hardware appliance which handles incoming and outgoing mail for an entire company. It also offers its services as virtual appliances and hosted services. The appliance approach has the advantage of allowing organizations to maintain control over their email systems.

The information on the Barracuda Networks' Spam and Virus Firewall website mentions the product provides anti-spam, anti-virus, anti-spoofing, anti-phishing anti-spyware, denial of service, and data leak protection through "twelve defense layers: network denial of service protection, rate control, IP reputation analysis, sender authentication, recipient verification, virus scanning, policy (user-specified rules), spam fingerprint check, intent analysis, image analysis, bayesian analysis, and rule-based scoring. One key sale point is its simplicity, promising easy installation and management. However, this is about as much information they provide about their methodology to the general public.

2.6.2 MessageLabs

MessageLabs (now a division of Symantec) is a UK-based company providing spam filtering, email virus protection, and encryption solutions. Their emphasis is to become a worry-free solution. If a company contracts their services, they will take care of everything. MessageLabs will monitor the performance of their filters to ensure their clients get what they paid for. Currently, the product website advertises 99% accuracy with less than 0.0003% false positive rate. In order to achieve their high accuracy, Andrew Oakley, anti-spam technical architect of MessageLabs, said they employ many employees to constantly monitor and modify their filtering technology as necessary. In their view, man is the best filter.

Organizations deploying MessageLabs' services do so by pointing all their MX records to MessageLabs. From there, MessageLabs dynamically filters spam and viruses and forwards legitimate email to the organization's mail servers. That way organizations can deploy whatever email management solutions suit their needs. Their services are considered premium, and therefore are not cheap. As with most commercial solutions, the details of their technology are a trade secret.

2.6.3 Google Gmail

With the introduction of Gmail, Google has also entered the spam filtering market. From its beginnings, Gmail has boasted an impressive spam filter which remains mostly secret.

Gmail offers complete cloud-based email solutions. Email is stored on their servers and all appropriate MX, and SPF records point to their servers. Users then connect to Gmail to send and retrieve email. Gmail is a complete email solution compared to Barracuda Networks or MessageLabs which still require the organization to manage their storage and delivery. This full approach, combined with a lower cost, makes Gmail a very attractive solution.

When it comes to spam filtering, however, Gmail has one weakness. Since there is little separation between the free, personal Gmail service and the commercial services they offer, spammers are able to test their latest assaults on Gmail address they control. Once they are able to successfully penetrate their filters, they can unleash their campaigns on all Gmail users, private and commercial. This problem is not unique to Gmail however. Yahoo!, Hotmail and other large email provides have the same problem.

2.6.4 MailRoute

MailRoute is another company providing high quality, low cost spam filtering. Their solution works by rerouting your email through their service, similar to the way MessageLabs does. The MailRoute website describes the product as a “multi-layered spam filtering system that identifies up to 96% of inbound spam” and “acheives an extremely low false-positive rate - 1 in 250,000 messages.” The website also states MailRoute’s multi-layered approach uses greylisting, blacklists, fingerprint databases, lexical analysis, bayesian filtering, distributed traffic analysis and user configured whitelists and blacklists.

2.7 Conclusions on the Literature Review

The arrival and growth of spam can be attributed to several factors: loose standards made worse by a lose implementation of the standards, a trusting community, and a group of people willing to move marketing into new areas. Much has been done about spam throughout the nearly 18 years of uncontrolled growth. Some solutions did not work, while some have stuck around for a long time. Companies have proved it is possible to get very high accuracy and precision, but the academic world doesn’t really know how.

Ultimately, we see that in spite of our efforts to end spam, spammer continue to evolve, adapt and bombard the world. So even if it seems the anti-spam world has the upper hand, the battle continues.

3 RESEARCH PROCEDURES

3.1 Finding Room for Improvement

Several researchers have suggested different approaches for filtering spam. However, there is little agreement on how to deal effectively with phishing attacks. Several sources hint that text classifiers are not up to the task. Putting together the pieces from the literature, several things become apparent:

1. The majority of the free and/or open spam content filters are based on Naive Bayes (generative model).
2. Naive Bayes classifiers are susceptible to performance degradation when the documents share many common features.
3. Creating a third category for phish may help isolate the features which may be unique to phishing email and therefore improve performance.
4. Non-generative models often achieve better performance than generative models since their training algorithms will ignore common features and focus only on the key differences between the documents. Non-generative models may therefore deal better with phishing attacks.

The literature contains much information about using non-generative text classifiers to identify and eliminate spam. However, missing from the literature were attempts of using a three-way (or three-class) classifier to classify emails into ham, spam and phish. Creating and testing this three-way classifier is the focus and main hypothesis of this thesis.

3.2 Overall Approach

To test the hypothesis, it will be necessary to put together a labeled email corpus that contains legitimate emails (ham), phishing emails (phish) and unwanted commercial email (spam). Once the corpus is built, a feature extractor will extract generic features from each of the emails and put them in a format which can be used by a freely available tool to train and classify the corpora. Once the classifier can be trained and validated, it will be necessary to adjust any parameters it may have to ensure the best possible classification performance.

Once a solid set of features and classifier parameters are found, a 10-way cross-validation will be run to test the entire corpus (see Section 3.5 for details) to find the filtering performance of the text classifier. First, performance will be measured with two categories (ham and spam), then with three categories (ham, spam and phish). Once those numbers are found, it will be possible to determine if using three categories improved phish detection without impacting the filtering accuracy of spam and ham.

3.3 Building the Necessary Corpora

There are three main corpora which are used research because they contain both legitimate and junk emails. The first is the SpamAssassing corpus. This is a labeled corpus (labeled meaning messages have been separated into ham and spam). The corpus is further divided into two parts: easy and hard. When combined, the corpus contains 6,951 ham messages and 2,398 spam messages. The ham portion contain some personal emails, but many are subscription-type messages with daily or weekly deals. Some may consider these messages unwanted. However, since these emails came from valid subscriptions, they are technically wanted and are not spam.

The second labeled corpus is the Ling-Spam corpus (Androutsopoulos, 2000). This corpus comes from the Linguist mailing list. The list is considered public, so publishing the emails does not create a security problem. However, in order to provide some level of security to its subscribers, some information has been removed. Additionally, the messages come pre-parsed and formatted for NLP tools and do not conform to email message standards. This corpus is therefore ideal for testing language tools, but inadequate for analyzing email structure information.

The third labeled corpus is the 2005 TREC Public Spam Corpus. It contains 92,189 total messages (39,399 are ham and 52,790 are spam). The messages come from Enron company emails made public by the Federal Energy Regulatory Commission during its trials. The headers contain very little modification and the messages contain Enron business emails without attachments (TREC, 2005).

Finally, Jose Nazario has collected phishing emails for several years and made them available, with slight header modification. There are four compilations total. The newest compilation has phishing emails from August 2006 to August 2007 and contains 2,279 phishing emails (Nazario, 2007).

3.4 Feature Extraction

Feature extraction refers to the process of extracting data from the documents which can help the classifier identify the document. There are many different ways to extract features for different applications. For some classification problems, extracting certain keywords is enough (Joachims, 2002). For other problems, all words and perhaps even word pairs are extracted. For problems where the document only contains a couple of words, characters are extracted to form *character distributions* (NLTK Documentation).

Kanaris and several others have suggested that using character distributions in the form of character-based n-grams yields excellent performance (Kanaris, 2006). In this approach *all characters* are used including white space, punctuation and control characters. Table 3-1 shows the features that would be extracted from the phrase *Hello World!* using single characters (unigrams), character pairs (bigrams) and character triplets (trigrams).

Table 3-1: Sample Character N-grams

N-gram size	**Features**
Unigram	`H e l l o [sp] W o r l d !`
Bigram	`He el ll lo o[sp] [sp]W Wo or rl ld d!`
Trigram	`Hel ell llo lo[sp] o[sp]W [sp]Wo Wor orl rld ld!`

Character-based n-grams provide several benefits over word-based n-grams. Kanaris has noted:

> Character n-grams are able to capture information on various levels: lexical ('the ', 'free'), word-class ('ed ', 'ing '), structural ('!!!', 'f.r.'). In addition, they are robust to grammatical errors and strange usage of abbreviations, punctuation marks etc. The bag of character n-grams representation is language-independent and does not require any text preprocessing (tokenizer, lemmatizer, or other 'deep' NLP tools). It has already been used in several tasks including language identification, authorship attribution, and topic-based text categorization with remarkable results in comparison to word-based representations.
>
> An important characteristic of the n-grams on the character-level is that it avoids (at least to a great extent) the problem of sparse data that arises when using n-grams on the word level. That is, there is much less character combinations than

word combinations, therefore, less n-grams will have zero frequency (Kanaris, 2006).

Additional considerations for the use of character-based n-grams are:

- They deal better with obfuscation, such as writing Viagra as `\/|@6R/-\` or using HTML tables to obfuscate words
- They work with all languages, even glyph-based without special modifications
- They can detect syntax styles in the way the email, HTML or Javascript is written

It is also important to note email messages have structure which can be used during feature extraction. For example, it is possible to construct different character distributions for the body of the email and the subject heading. It is also possible to use other headers such as the To, From, Return-Path and/or Received headers which are often forged by spammers.

Using character n-grams over words may seem unintuitive and perhaps even inefficient at first. To understand their value we have to remember that spam filtering is not a typical classification problem. Unlike other classification problems (such as finding sports or political news in a news feed), spammers are often trying to evade proper classification. Take, for example, the following subject from a spam message, taking notice of the space between letters and the hyphens between words:

```
Y o u r - A D - on 2 - M i l l i o n - W e b s i t e s
```

A word-based feature extractor would extract each individual letter and not make much sense of them. A character-based unigram feature extractor would extract the same letters if the spaces were not there and notice key words like “AD.” Additionally, the feature extractor would note a very high number of spaces which may rarely happen in legitimate messages. Thus the features given to the classifier capture word-equivalent information plus additional information to aid in the classification process.

Consider another example: the email's *From* field. Character distributions are very capable of discerning minute differences in personal names. An email with a sender of *p4ZogeKa3Mahoney* looks very different statistically to *John Smith.* In fact, character-based n-grams are so sensitive they can even distinguish gender (NLTK Documentation, n.d.).

Best of all, character-based features achieve all of these benefits by simply collecting individual and groups of characters. There is no need for anti-obfuscation measures or lexical analyzers to improve feature extraction.

During the implementation of the feature extractor it will be necessary to test its performance and find the best combination of features which can produce respectable classification performance without requiring an immense amount of computing power or storage. Respectable performance will be defined as accuracy greater of 95%. During this process it will be necessary to employ *Feature Engineering* to improve classifier performance. Feature engineering is an iterative process which includes feature design, selection, induction and impact analysis. (Ringger, 2005). As a rule, however, all features remain generic. That is, no features match specific characteristics of the corpora, such as specific phrases, senders, recipients, subjects or even words.

Extracted features are stored in the SVMlight file format. This format combines all documents into one file. Documents are represented individually, one per line, with a numeric class identifier first, followed by the features separated by spaces. Features are represented in pairs delimited by colons. The first value in the pair is the feature number, the second is the number of times the feature appeared in the document. It is possible to skip feature numbers for features which were not present in the document (SVMlight Documentation, n.d.).

3.5 Training and Classification

In order to avoid the bias that may come from the selection of training and validation groups, a ten-way cross-validation will be used in all training and classification runs. The cross-validation is done by dividing the individual corpora into ten equal sections. Then, nine of the ten parts are chosen to serve as the training corpus, and the tenth part is used for validation and performance measurements. Ten iterations are performed, ensuring each of the ten groups is used as the validation group only once. The raw performance numbers are then added together, allowing us to classify the entire corpus (Kanaris, 2006).

3.6 Finding a Classifier

As mentioned in Section 3.1, non-generative classifiers offer excellent performance when classifying spam. Some non-generative classifiers include Maximum Entropy (MaxEnt), Support Vector Machines (SVM), Decision Trees and Neural Networks. Maximum Entropy and Support Vector Machines will be considered here as their use is well established in the literature and their high performance (Zhang, 2004; Kanaris, 2006).

3.6.1 Maximum Entropy

Maximum Entropy is a model which tries to find a distribution with the highest entropy. Entropy, in information theory, “measures the amount of information in a random variable” (Manning, 2000). Manning and Schütze define Maximum Entropy modeling as follows:

> Maximum entropy modeling is a framework for integrating information from many heterogeneous information sources for classification. The data for a classification problem is described as a (potentially large) number of features. These features can be quite complex and allow the experimenter to make user of prior knowledge about what types of information are expected to be important for classification. Each feature corresponds to a constraint on the model. We then compute the *maximum entropy model,* the model with maximum entropy of all models that satisfy the constraints. . . . Choosing the maximum entropy model is motivated by the desire to preserve as much uncertainty as possible. . . .
>
> In maximum entropy modeling, feature selection and training are usually integrated. Ideally, this enables us to specify all potentially relevant information at the beginning, and then to let the training procedure worry about how to come up with the best model for classification (Manning, 2000).

A publicly available NLP framework with Maximum Entropy support is the Natural Language Toolkit (NLTK). This framework is written in Python. Unfortunately, preliminary testing proved it to be slow and incapable of handling large feature sets required for this application.

Another publicly available toolkit is Mallet (MAchine Learning for LanguagE Toolkit), courtesy of the University of Massachusetts at Amherst (McCallum, 2002). This toolkit has the advantage that it is ready run and only requires data. It even includes a simple word-based feature extractor and a tool to convert SVMlight data to its own vector format.

3.6.2 Support Vector Machines

Support Vector Machines attempt to find a hyperplane that can separate two distinct sets of documents. Joachims explains it as follows:

> Let's assume that the training data can be separated by at least one hyperplane h'. This means that there is a weight vector w and a threshold b', so that all positive training examples are on one side of the hyperplane, while negative training examples lie on the other side (Joachims, 2002).

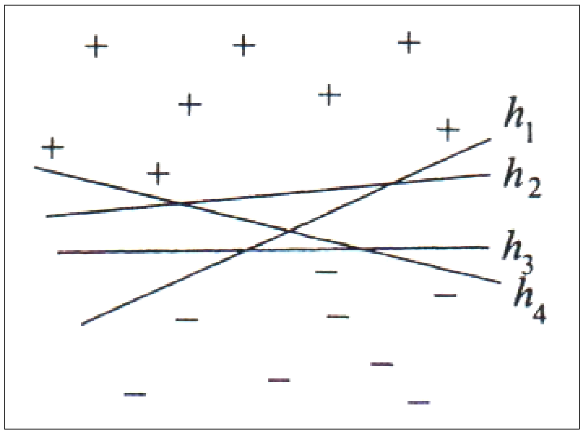

Figure 3-1: SVM Hyperplane Classification

> In general, there can be multiple hyperplanes that separate the training data without error. . . . From these separating hyperplanes, the support vector machine chooses the one with the largest margin δ. (Joachims, 2002).

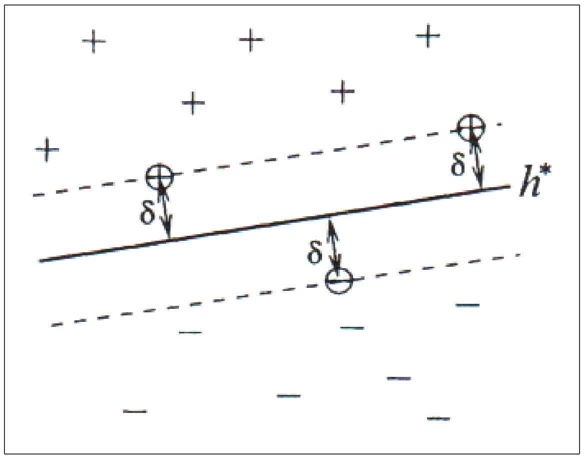

Figure 3-2: SVM Margin and the Hyperplane

The simplest hyperplane that can divide the two sets of documents is a straight line, either with a hard margin (as explained in Figure 3-2) or with a soft margin where distance from the support vectors and the hyperplane are different (Joachims, 2002). Some real-world data sets don't fit into linear models. SVMs allow for non-linear planes through polynomial hyperplanes or the use of a kernel functions to map a non-linear space into an equivalent linear space (Joachims, 2002).

One drawback of SVMs is that they only consider *binary* classification with positive and negative documents. Fortunately, it is possible to do multi-cateogry classification with SVMs. One way is through "reducing a single multiclass problems into multiple binary problems" (Crammer, 2001) This solution is considered suboptimal since "it cannot capture correlations between the different classes since it breaks a multiclass problem into multiple *independent* binary problems" (Crammer, 2001). Crammer and Singer instead suggested a different approach in 2001 with a "simple generalization of separating hyperplanes and, analogously, a generalized notion of margins for multiclass problems" (Crammer, 2001).

SVMlight is considered the reference implementation for Support Vector Machines text classifiers. Two versions are available: SVMlight for traditional binary classification and SVM Multiclass which uses the Crammer and Singer method for multi-class classification. The default kernel in SVMlight and SVM Multiclass are different. The SVM Multiclass kernel has the advantage of being much faster than the default SVMlight kernel.

3.6.3 Choosing the Classifier

Initial work for this research began with Mallet. It offered several advantages: it provided Maximum Entropy modeling, automatic 10-way cross-validation, and offered excellent speed. However, it had one drawback: the maximization algorithm would error out with large feature sets and deliver suboptimal results and had to be scrapped. This forces the use SVM Multiclass for the classifier. One advantage of using SVM Multiclass is that features will have to be extracted to svmlight files which can also be used by Mallet. (Mallet vector files are not compatible with SVM Multiclass, however.)

3.7 Measuring Performance

To test the performance of the two and three-way classifiers raw data will be put into confusion matrices (see section 3.7.1). From these matrices accuracy, precision and recall are calculated as stated in the formulas 3-1, 3-2 and 3-3. Precision and recall can be calculated separately for ham, spam and phish. For the purposes of measuring performance, precision and recall will be calculated for ham only, unless otherwise stated.

3.7.1 Confusion Matrix

A *confusion matrix* provides a way to summarize and tabulate classifier performance. For this thesis, truth will be represented in rows, while the prediction of the classifier will be identified by the column. For the two-class configuration, the confusion matrix will be constructed as shown in Table 3-2, where $n_{h \to h}$ represents the number of ham messages classified as ham, $n_{h \to s}$ represents the number ham messages incorrectly classified as spam, $n_{s \to h}$ represents the number of spam messages incorrectly classified as ham, and $n_{s \to s}$ represents the number of spam messages classified as spam (Kanaris, 2006).

Table 3-2: Binary Confusion Matrix

	Ham	**Spam**
Ham	$n_{h \to h}$	$n_{h \to s}$
Spam	$n_{s \to h}$	$n_{s \to s}$

In the binary case, categories are often described as positive and negative (see Section 3.6.2). In this thesis, ham messages are considered positive, while spam and phish messages are considered negative. True positives (*tp*) are positive messages that were properly classified. Consequently, true negatives (*tn*) are negative messages that were properly classified. False positives (*fp*) are positive messages that were incorrectly classified (in our case, ham messages classified as spam) and false negatives (*fn*) are negative messages that were incorrectly classified (see Table 3-3).

Table 3-3: Binary Confusion Matrix with Alternate Terminology

	Ham	**Spam**
Ham	*tp*	*fp*
Spam	*fn*	*tn*

3.7.2 Accuracy

Accuracy is measured by taking the number of messages that were correctly classified (whether positive or negative) divided by the total number of messages (Manning, 2000):

$$accuracy = \frac{tp + tn}{tp + fp + fn + tn} \qquad (3\text{-}1)$$

For the three-category classification, spam and phish messages will be considered as correctly classified if they are *not* classified as ham (that is, classified as either spam or phish). This approach is acceptable since a spam filter would throw out any emails labeled as either spam and phish.

From the accuracy we can also obtain the *error rate* which is simply 1 minus the accuracy.

3.7.3 Precision

Ham precision is calculated by taking the number of ham messages properly identified as ham (true positives), divided by the total number of messages identified as ham (Kanaris, 2006):

$$precision=\frac{tp}{tp+fn} \tag{3-2}$$

Precision measures how many of the messages classified as ham were actually ham, providing an indication of filter safety (Kanaris, 2006). This measurement can also be computed in terms of spam or phish.

3.7.4 Recall

Recall is computed by taking the number of ham messages properly identified as ham divided by the total number of ham messages:

$$recall=\frac{tp}{tp+fp} \tag{3-3}$$

In simple terms, recall measures how many of the known ham messages were correctly classified. This measurement can be considered the effectiveness of the classifier (Kanaris, 2006). This measurement can also be computed in terms of spam or phish.

From the recall we can also obtain the false-positive rate, which is simply 1 minus the recall.

3.7.5 Student's *t*-test, *p*-value and Statistical Significance

The student's t-test provides a statistical method to "estimate error" around a mean. When combined with the p-value, it allows for testing "the difference between means" (Ramsey, 2002). The p-value provides the probability that the two means (and their corresponding distributions) are equal. "If the *p*-value is small, then either the hypothesis [that the means are equal] is correct —and the sample happened to be of those rare ones that produce such an unusual [error]–or the hypothesis is incorrect. . . . The *p*-value . . . therefore provides a measure of credibility" (Ramsey, 2002). Thus, the *p*-value can be considered as the probability that the two means are *not* equal. In statistics, a *statistically significant* result is one where we can say with 95% certainty that it is true. Consequently, in order to determine with 95% confidence that two means (or in this case, two classification results) are different, we need a corresponding *p*-value of 0.05 from the two different results. The computation of the *p*-value is quite complex. Fortunately, statistical software and several websites make it really easy to compute from two sets of data points.

The 10-way cross-validation explained in Section 3.5 provides the perfect foundation to statistically verify the difference between the two-way and three-way classifiers. By taking the accuracy of each of the 10 classification runs and providing them to a *t*-test calculator, I will be able to determine with 95% confidence if the three-way classifier results are different from the two-way classifier and therefore statistically significant.

4 RESULTS AND DATA ANALYSIS

4.1 Building the Corpora

The corpora was built from the SpamAssassin spam corpus which contains both spam and ham. For the phish corpus Jose Nazario's third phishing corpus was used. (See Section 3.3) The total number of messages on each of the corpora is given in Table 4-1.

Table 4-1: Number of Messages in Corpora

Corpus	Number of Messages	Wanted/Unwanted Totals
Ham	6,951	6,951
Phish	2,279	4,677
Spam	2,398	

4.1.1 Splitting the Corpora

As explained in Section 3.2, each corpus needs to be split into 10 equal parts in order to do the 10-way cross-validation. Since none of the message counts are exactly divisible by 10, it was necessary to find a mechanism that would ensure all 10 parts would be as balanced as possible. (An unbalanced approach would be to divide the phish corpus into 9 parts of 227 with the 10th part having 236 messages. Or, divide it into 9 parts of 228 with the 10th only having 227 messages.)

To split the corpora evenly, I devised a simple rounding mechanism to determine where messages should be placed. Here is an example. Let's assume we have 16 messages which are going to be divided into 6 groups. Dividing the number of messages by the number of groups would yield 2.667 messages per group in this case. Since it is not possible to divide individual elements, traditional rounding is used to find the number of elements in each group. Group 1 should contain documents 1 through 2.667. Rounding 2.667 to 3 means documents 1, 2 and 3 are put into Group 1. The next range should contain the next document (4 in this case) through the next division point: 5.333. Rounding 5.333 down to 5 means Group 2 contains documents 4 and 5. The next division point is 8, so Group 3 contains documents 6, 7 and 8. These numbers are then repeated for Groups 4, 5 and 6 which contain 3, 2 and 3 documents respectively. With this technique, deficiencies and remainders are divided equally and are balanced among all the groups. Table 4-2 lists the groups and the number of messages from each corpus.

Table 4-2: Split Corpora Group Breakdown

Group	**Number of Messages**		
	Ham	**Phish**	**Spam**
01	696	227	240
02	695	228	240
03	695	228	239
04	695	228	240
05	695	228	240
06	695	228	240
07	695	228	240
08	695	228	239
09	695	228	240
10	695	228	240

One more issue remains: which messages should go into each group. For the ham and spam corpora from SpamAssassin, messages were stored one per file. The messages were divided by sorting them by their filename in a case-sensitive alphabetic order (normal output of `ls` command in Unix). The messages were then sequentially renamed to their ordered sequence number with leading zeros (0000, 0001, 0002, 0003, etc.) Then, as listed on Table 4-2, the appropriate number of messages was placed into each group.

For the phish corpus the messages come in the mbox format (all emails in one file). To separate the emails into separate files, it was necessary to import the mbox file into an email program (in this case, KMail 1.7.x). From there, the messages were moved to a maildir mailbox and the resulting individual files were divided in the same way the ham and spam corpora. It is assumed the messages were separated in the order they appeared in the mbox file.

4.2 Extracting Features

There are two types of features that were used: continuous and binary. Continuous features count the number of times a particular feature is found. For example, for unigrams, it will count the number of times a particular character (such as a the letter A) is found. Another example of a continuous feature is the number of attachments an email may contain. Binary features don’t count the number of features found; they only record that a particular feature exists in the document. For example, whether the email contained attachments. The inspiration for the features will be explained as the features are created. (The source code for the Feature Extractor class and the associated executable, both written in Python 3, can be found in Appendix E.)

For all Feature Engineering iterations, the purpose was to create a respectable email text classifier/spam filter. Thus, a full 10-way cross-validation was performed using the ham and spam corpora. Phish was not included at this stage in order to focus on building a solid spam filter.

4.2.1 Unigrams, Bigrams and Trigrams

As explained and justified in Section 3.4, all text-based features will use character-based n-grams. Kinaris suggests n-grams of sizes 4, 5 and 6 yield the best classifier performance. However, to keep things simple, n-grams of sizes 1 (unigrams), 2 (bigrams) and 3 (trigrams) were used. Figure 4-1 shows this approach yields acceptable performance (over 95% accuracy) when using unigrams and bigrams, or unigrams, bigrams and trigrams on the body of the message.

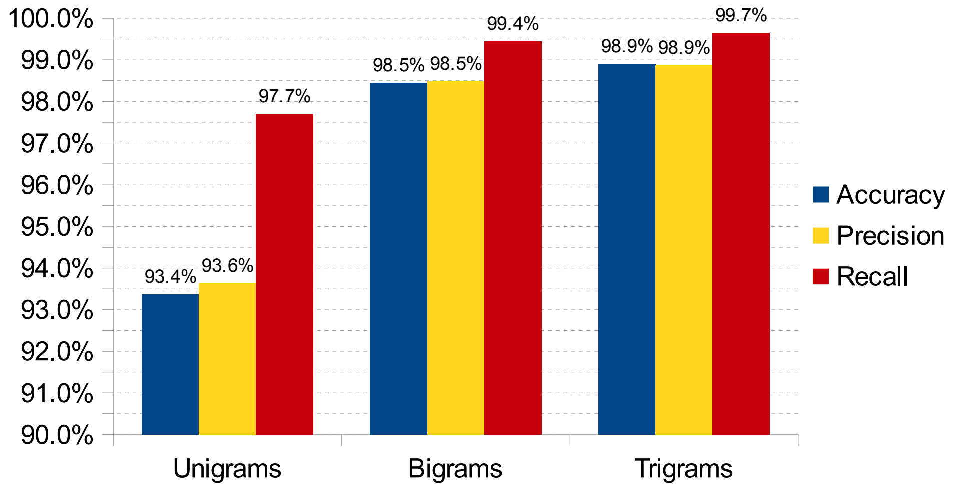

Figure 4-1: Performance Comparison of Unigram, Bigram and Trigram Distributions for Ham and Spam

4.2.2 Header Features

As suggested by Graham and Kanaris, email messages contain structure which can be used in addition to pure text classification (Graham, 2003). This structure can be represented as additional features. For example, a separate character distribution can be created for the characters in the subject, to/from headers, etc. Since it is often possible for humans to detect spam simply by looking at the email's subject, the subject header was added as its own distribution. Seeing as the to/from headers are often forged and the forgery is often evident to humans, to/from headers were also included, each as its own distribution. Another forged header rarely seen by users is the Return-Path (the email address presented during the MAIL-FROM SMTP command). This was added as well at the suggestion that the Return-Path yields effective results in detecting spam (Treviño, 2007a). After a quick run of these new header features and the errors incurred, a pattern was found that many of the misclassified ham messages came from mailing lists. Thus, the Mailing-List and List-Id headers were added. The performance if these new features, both by themselves and combined with the body is summarized in Figure 4-2.

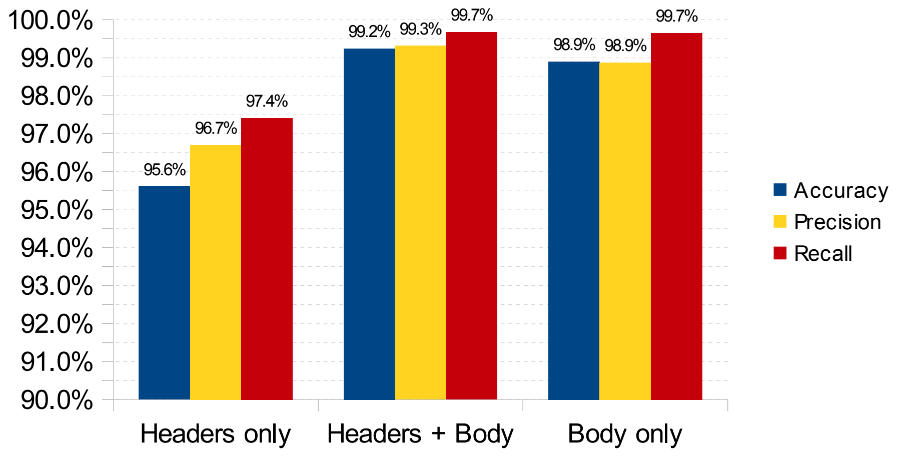

Figure 4-2: Performance Comparison of Features from Headers and Body

4.2.3 Splitting Text and HTML Message Parts

Continuing with the concept of structure, it seems logical to split the plain text and HTML versions of the email into separate distributions. This would help fingerprint different body parts separately and hopefully improve performance. Additionally, counting the number of plain text parts, HTML parts and other parts (such as attachments) seemed logical as well. This helps identify message that only contain plain text or HTML, or any other format combinations. Finally, while analyzing the classifier errors, I noticed some misclassified spam messages had both plain text and HTML parts, but the plain text part was completely empty. Another feature was added to specify a part had been defined in the multi-part message structure but did not have any real content in it.

The performance when these features are added is summarized in Figure 4-3. The blank-part feature helped eliminate spam messages with blank text parts. Unfortunately, adding the count feature did affect performance, and splitting the multi-part messages into separate distributions hurt performance slightly.

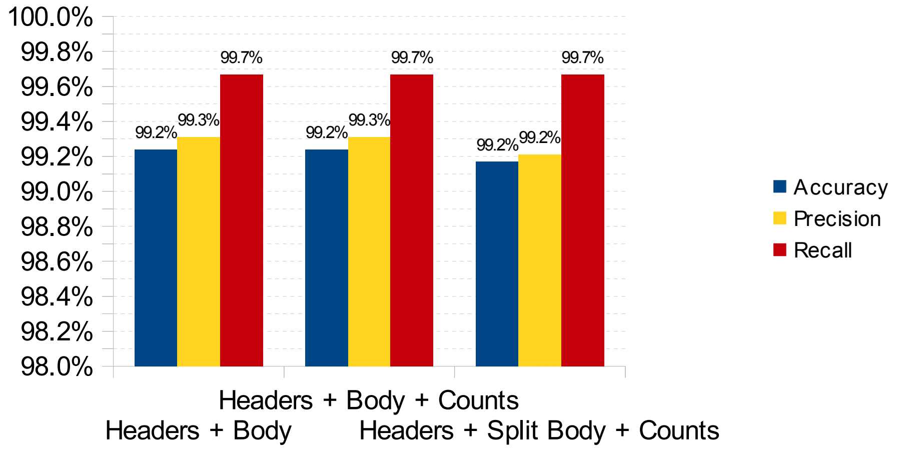

Figure 4-3: Performance with Multi-part Counts and Splitting Multi-part Messages

4.2.4 Using the Start and End of the Message Body

It has become apparent by this point in the feature extraction process that the more features are added the longer training takes. Humans seldom read every word in an email before determining it is unwanted. Thus, it should be possible to extract features only from the beginning and/or end of the message. I created an alternate feature set where only the first 1024 and last 1024 characters of text and HTML parts are used. For messages with less than 2048 characters, the entire text is used. The classification performance of the *short body* compared to the entire body of the message is summarized in Figure 4-4.

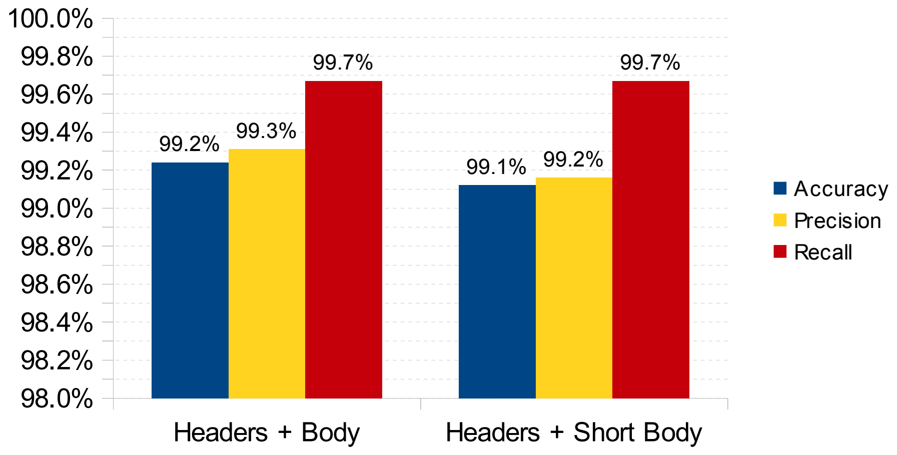

Figure 4-4: Performance Comparison Between Full and Short Body

Because the amount of data that is actually processed is reduced, short body features bring a noticeable improvement in speed, as seen in Figure 4-5.

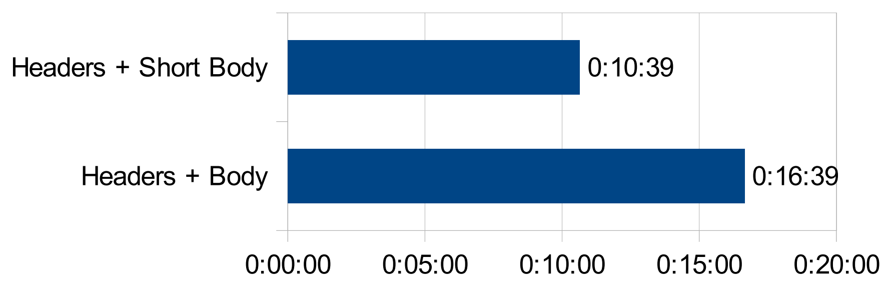

Figure 4-5: Execution Run Time Comparison Between Full and Short Body

Extracting features from the beginning and end of the body has the added benefit of providing basic content length normalization, ensuring long messages are not weighted more heavily than smaller messages. Joachims provides several mechanisms for normalizing continuous features (Joachims, 2002). SVMlight also provides kernels for these normalized feature sets. However, when used, training took considerably longer and performance dropped considerably. I wasn't sure why there was a drop in classification performance so feature normalization was abandoned.

4.2.5 Phish-Specific Features

Fette suggests 10 features which may help detect phishing emails which "[extract] information that can be used to detect deception" (Fette, 2007). Some of those features use DNS record analysis which is hard to implement after the fact. However, the URL analysis features they suggest appear to be quite useful. I programmed the feature extractor to look for four of these features in the HTML part of the messages. These were programmed as binary features and are described in Table 4-3.

Table 4-3: Binary URL Features

Feature	Description
Raw URL	The text of the URL is a URL (no human-readable link description): `<a href="http://www.google.com">http://www.google.com</a>`
Non-matching Raw URL	The text of the URL is a URL, but it does not match the href property of the anchor tag: `<a href="http://www.google.com">http://www.yahoo.com</a>`
"Click here" URL	The text of the URL contains variations of the phrase *click here:* `<a href="http://www.google.com">CLICK HERE</a>`
IP URL	The href property points to an IP address, suggesting a rouge site: `<a href="74.125.224.115">CLICK HERE</a>`

In addition to the binary features, an additional continuous feature was added: the number of domains referenced. Phishing attacks often link to the real site and to the rouge (fake) site set up by the attacker. (Fette, 2007).

The addition of the URL analysis features improved performance slightly, as summarized in Figure 4-6.

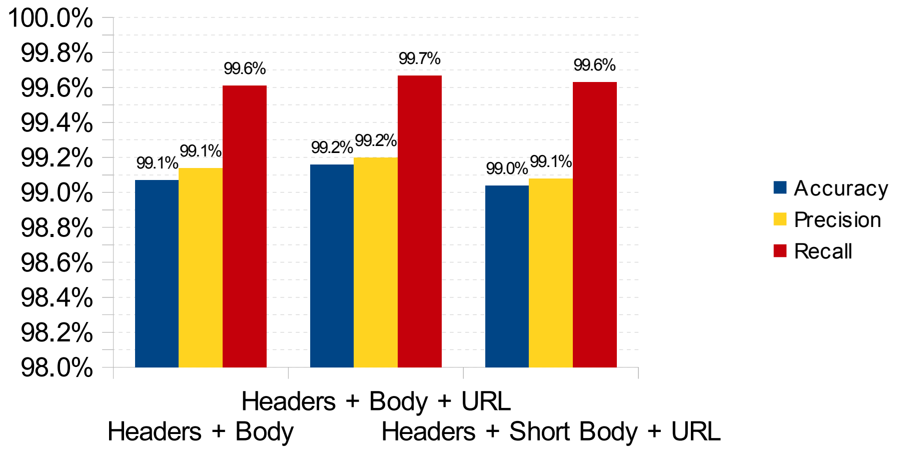

Figure 4-6: Performance of URL Features

4.2.6 Message Parsing Error Features

While writing the feature extractor Python required proper handling of message parsing errors. Assuming legitimate messages should not contain parsing errors but spam or phish might, I decided to add two binary features that would record these errors. The first feature records when Python is unable to parse the email message. The second records when Python encounters a character decoding error in the message. Python 3 is able to ignore the character decoding error allowing the feature extractor to extract the features from the text, the offending characters and that fact that offending characters existed in the message.

4.2.7 Corrections to the Corpora

During the feature engineering cycle there were four messages from the phishing corpus which were consistently categorized as ham. As it turns out, the classifier was right: three of these messages were clearly legitimate: 0118, 0768, and 0908. To correct this error, these messages were moved from the phishing corpus to the ham corpus. The form message was suspicious, because it was a phish message which had been forwarded to Jose Nazario (Figure 4-7).

Message 1352 posed an interesting dilemma: is the message actually unwanted? The original message certainly was. But, if a known recipient emails you an unwanted email (like in this case, for research) certainly this message is wanted. Additionally, the sender forwarded a text version of the message, removing the attack’s payload. Interestingly the classifier believed the message was wanted and constantly classified it as ham, which I believe to be correct. At first, moving it to the ham corpus seemed like the right thing to do. However, the author of the corpus believed that message to be phish. Keeping the message in the phish corpus would add an additional error to the results. Moving the message would introduce my bias into the corpus. The best and simplest solution I came up with was to simply throw the message away.

```
From: David Helder <dhelder@gizmolabs.org>
Subject: Weird phish
To: user <user@example.com>

For your collection.

Begin forwarded message:
> From: "eBay" <vote@eBay.com>
> Date: January 19, 2007 9:55:11 PM GMT-05:00
> To: dhelder@umich.edu
> Subject: Your opinion counts
>
> January 19, 2007
>
> Dear eBay Community:
>
> We have decided to close eBay on 27 February 2007 due to the
> repeatedly abuses on our company. We ask your opinion on this
> matter and we want to know if you agree with us or disagree .Below
> you can make your choice.
>
> If you want eBay to stay open click YES otherwise click NO .Your
> opinion is very important to us. If 50% of the eBay members vote
> positive eBay stays open otherwise it will be closed.
>
> Regards,
> eBay Team
>
>
> YES                                    NO
>
>
> eBay sent this message to you based on your good account standing.
> Your registered name is included to show this message originated
> from eBay.
>


--
David Helder - dhelder@gizmolabs.org - http://www.gizmolabs.org/~dhelder
```

Figure 4-7: Phish Message 1352

To avoid having to redo a lot of the work that had been done already, I decided to simply move the three mislabeled messages to their corresponding ham group. The forth problematic email was simply thrown out. However, I did not rebalance the numbers in each group, creating slightly imbalanced groups. Since the changes are small I did not feel this was detrimental to the analysis or results. In the interest of full disclosure Table 4-4 contains the final number of messages in each group. The numbers in bold represent the changes from table Table 4-2.

Table 4-4: Final Corpora Group Breakdown

Group	**Number of Messages**		
	Ham	**Phish**	**Spam**
01	696	227	240
02	695	228	240
03	695	228	239
04	**697**	**226**	240
05	**696**	228	240
06	695	**226**	240
07	695	228	240
08	695	228	239
09	695	228	240
10	695	228	240

4.2.8 Feature Summary

Appendix A contains a full list of the feature sets which can be extracted by the feature extractor. Each feature set consists of one or more of the features described in Section 4.2. Each feature set was given a descriptive name which is used throughout the text and it’s included in the table.

The best feature combinations use headers and a single n-gram distribution of the message body, with the addition of the HTML text (stripped of tags). Using the shortened body as described in Section 4.2.4 yields very similar performance, but greatly reduces computational running time. Because of their performance, classifier performance comparisons will use variations of these features.

4.3 Maximizing SVM Performance

SVMlight includes a parameter which specifies "the tradeoff between training error and margin" (SVMmulticlass). In short, the higher the number, the better filter performance at the expense of training time. To maximize filter performance it is necessary to find the right value.

To find the optimum value of *c* I ran a series of tests on some of the best feature sets with different values of the tradeoff value *c*. The range of values tested was 0.1, 1, 10, 100, 500, 1,000, 5,000 and 10,000. The accuracy of the filter at each value of *c* was recorded along with its total run time (training + classification) for the 10-way cross-validation. The performance peaked at around 1,000. Values of *c* above 1,000 rarely yielded better results. For many of the feature sets, the best performance was achieved with a value of 500. (The full results are found in Appendix B.) Figure 4-8 summarizes the classification performance for the various values of *c*, while Figure 4-9 summarizes the total running time which increases proportionally to the classifier performance.

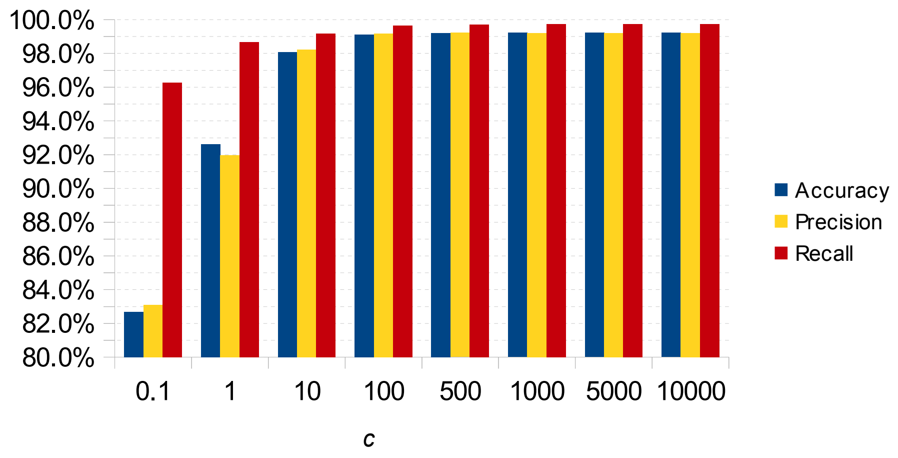

Figure 4-8: Classification Performance for Various Values of *c*

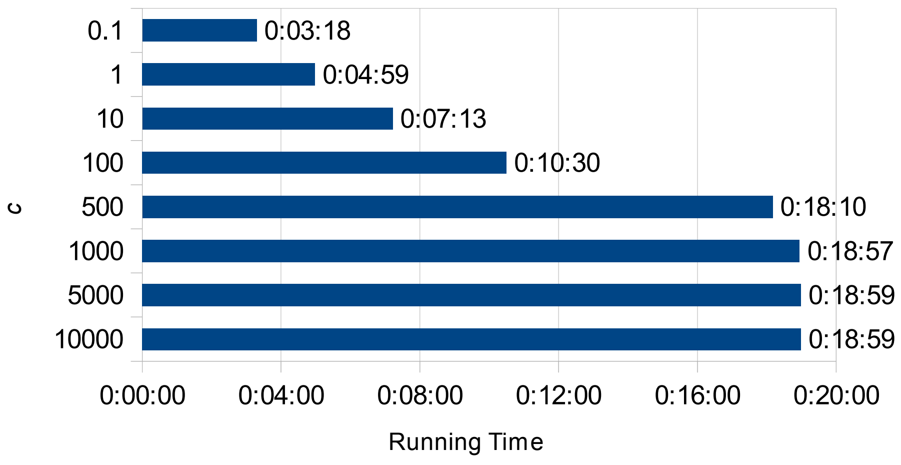

Figure 4-9: Running Time for Various Values of *c*

4.4 Confirming the Phishing Problem

With several high-performance feature sets and a tuned classifier, it is possible to test the phish corpus. The first test classifies the entire phish corpus with a classifier model trained entirely from the ham and spam corpora and no phish. By blindly classifying phishing emails without any prior knowledge, we will be able to quantify how the classifier behaves. If phishing emails truly resemble legitimate email more than spam, we should see most of the phish corpus classified as ham and consequently, achieve less than 50% accuracy. If the classifier is truly confused, accuracy (in theory) should be around 50%. If the classifier believes most of these messages are unwanted, accuracy will be above 50%. Figure Figure 4-10 shows the results of this first test with feature sets that did not use Fette’s URL features. Figure 4-11 Shows the results with the URL features. Appendix B contains the full results.

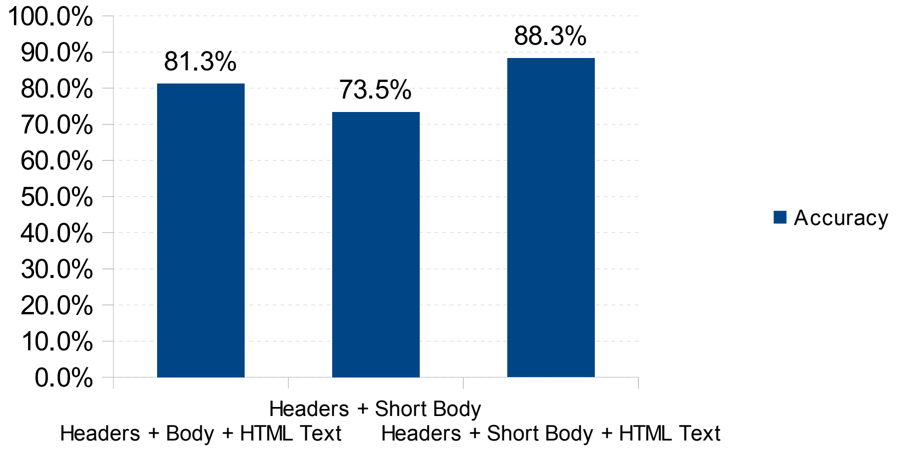

Figure 4-10: Blind Phish Classification Results

These results are surprising. The data suggests that the phishing emails in the phish corpus look more like spam. The range and confidence interval for the data leads to the conclusion (with 95% confidence) that phish resembles spam much more than it does ham (see Figure 4-12).

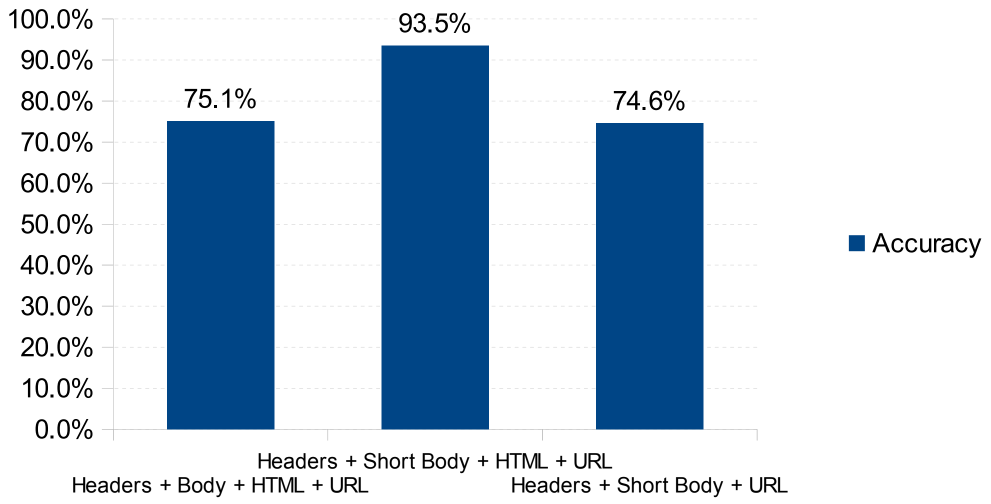

Figure 4-11: Blind Phish Classification with URL Features

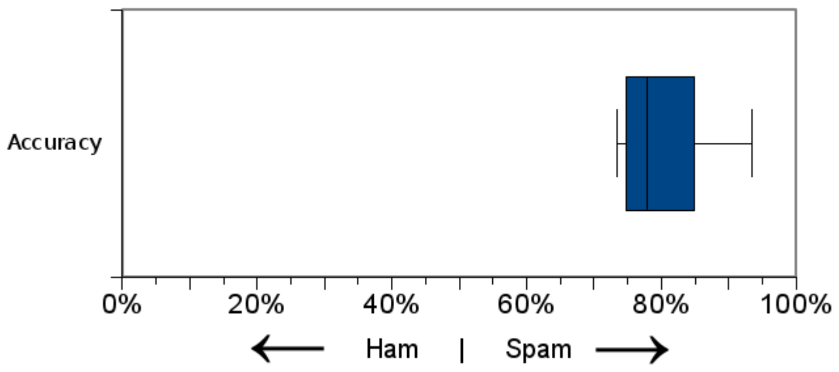

Figure 4-12: Accuracy Range and Skew of Blind Phish Classifier

4.5 Results of the Two-Way Classifier

The next test establishes the baseline performance to test my hypothesis. For this test the phishing corpus will be added to the spam corpus to train a model with a traditional two-class configuration. Figure Figure 4-13 provides a summary of the results.

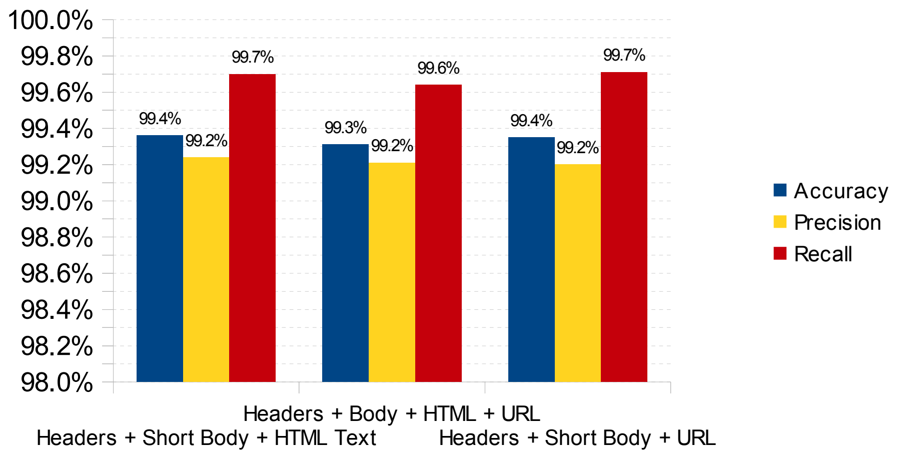

Figure 4-13: Classifier Performance for Ham, Spam and Phish in Traditional 2-Way Configuration

The baseline performance remained consistent with the original performance without the phishing emails. The data in Appendix B suggests there was no statistically significant drop in performance. Figure Figure 4-14 shows the performance of the *Headers + Short Body + URL* feature set with and without phishing emails side by side.

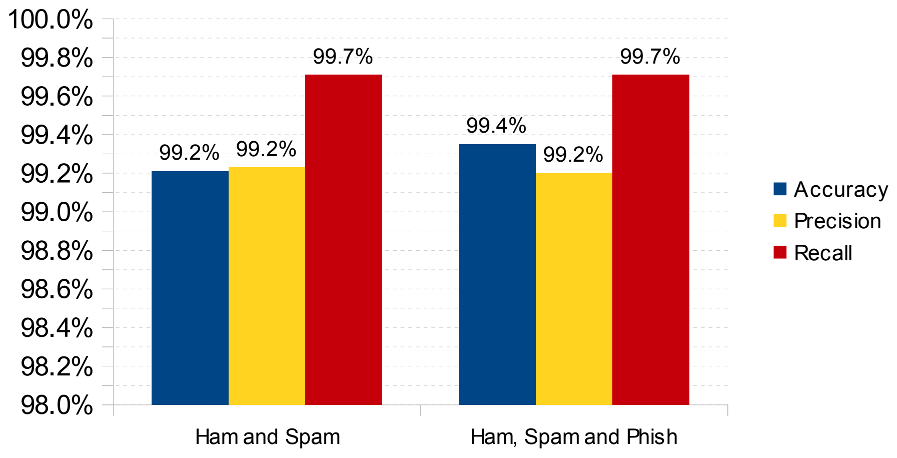

Figure 4-14: Comparison of Classifier Performance With and Without Phish Corpus

Comparing the two results shows comparable numbers with a slight increase in accuracy when the phish corpus was introduced. Table 4-5 Provides the confusion matrix and class-specific recall, showing no significant changes in performance.

Table 4-5. Classifier Performance Comparison in a Two-Class Configuration

Ham and Spam			
	Ham	**Spam**	**Recall**
Ham	6934	20	99.71%
Spam	54	2344	97.75%

Ham, Spam and Phish			
	Ham	**Spam**	**Recall**
Ham	6933	21	99.70%
Spam	49	2349	97.96%
Phish	6	2263	99.74%

4.6 Results of the Three-Way Classifier

The next step is to test the performance of the classifier in the three-class configuration. The results are summarized in Figure 4-15 and compared to the two-class configuration in figure Figure 4-16.

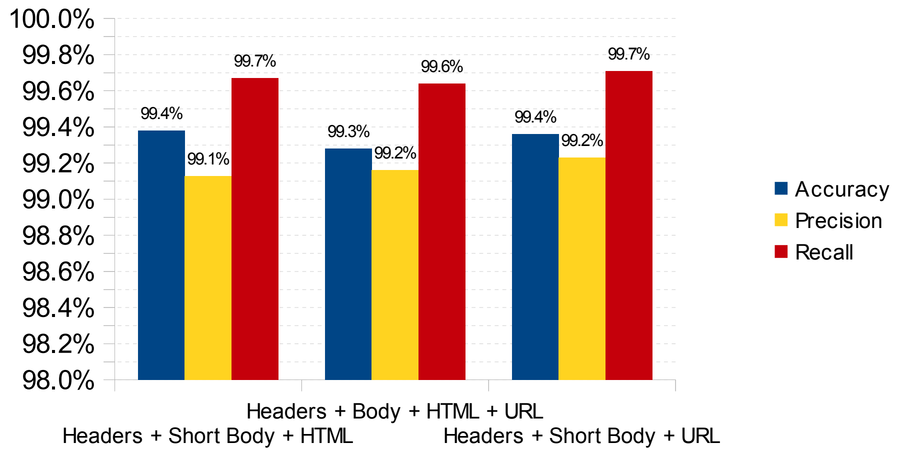

Figure 4-15: Performance of the Three-Way Classifier (*c* = 500)

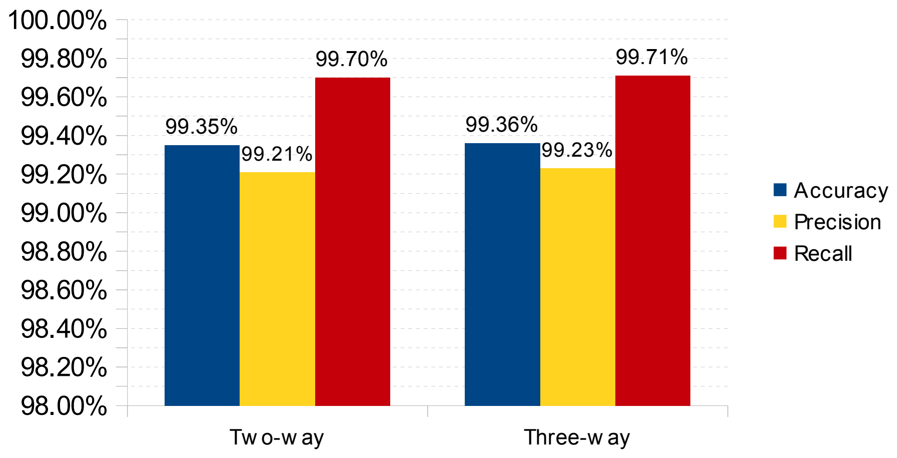

Figure 4-16: Performance of Two-way vs. Three-way Classifiers, (Headers + Short Body + URL, *c* = 500)

These results are quite intriguing. Accuracy improved 0.01% while precision and recall improved by 0.02% and 0.01% respectively. Table 4-6 contains the raw data along with the recall for spam and phish while Figure 4-17 Shows runtime nearly tripled with the three-way classifier.

Table 4-6: Performance Comparison of Two-way vs. Three-way Classifiers with Class-specific Recall (*c* = 500)

Two-way Classifier			
	Ham	**Spam**	**Recall**
Ham	6934	20	99.71%
Spam	49	2349	97.96%
Phish	3	2272	99.87%

Three-way Classifier				
	Ham	**Spam**	**Phish**	**Recall**
Ham	6934	14	6	99.71%
Spam	50	2344	4	97.75%
Phish	4	21	2250	98.90%

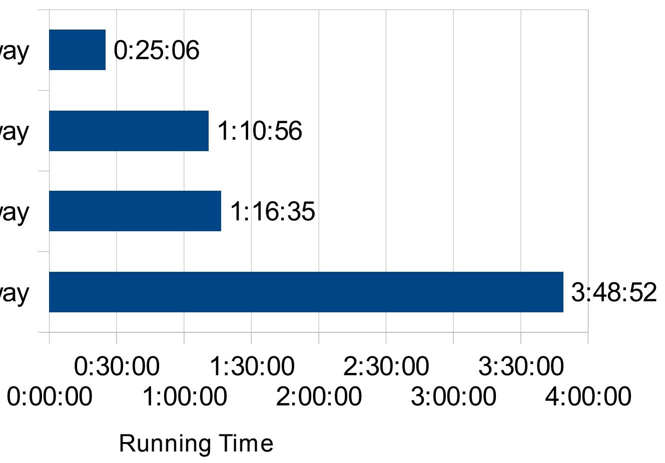

Figure 4-17: Running Time Comparison Between Two-way and Three-way Classification

The result of the t-test on the data allows us to see if the differences in performance are statistically significant. As given in Appendix D, the *p*-value comparing the two results is 0.48, which means the differences in performance are statistically *insignificant.* This means **there is no significant performance to be gained by using a three-way classifier** (see Figure 4-18). The original hypothesis that a three-way classifier will improve phishing detection is proved to be incorrect.

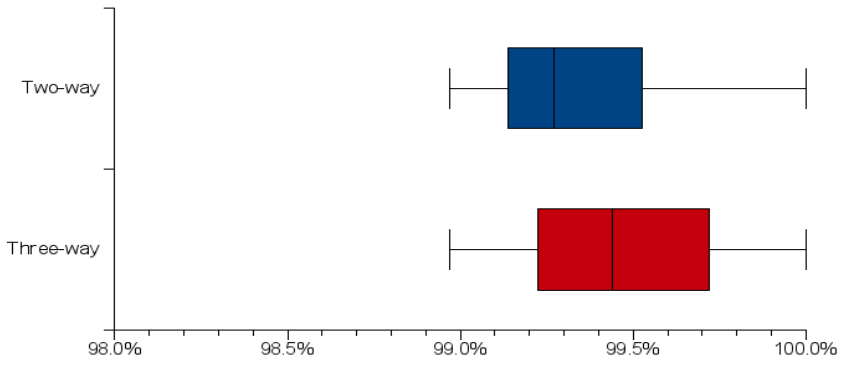

Figure 4-18: Comparison of Mean and Variance of the Accuracy of Two-way vs. Three-way Classifier

4.7 Further Analysis of the Results

Another way of explaining the results is that the three-way classifier was not able to beat the traditional two-way classifier because a two-way classifier already achieves similar performance. This contradicts the literature which suggests content filters can't deal with phishing emails as efficiently as regular spam (Fette 2007, 1). In fact, the data in Table 4-6 suggests the classifier was able to differentiate phish better than spam. This discrepancy is something I explored further to try to find the reason why other researchers reached their conclusions.

Before trying to explain the discrepancy, I must first offer a preliminary conclusion: even though phishing emails are designed to appear legitimate, they are not so legitimate that text classifiers have a hard time distinguishing them. With proper training, non-generative text classifiers are capable of finding the differences between phishing and legitimate emails enough to tell them apart. This is the perspective I used to explain the discrepancy between my findings and those found in the literature.

4.7.1 Corpora

There is a possibility the corpora itself may introduce features that help the classifier discriminate phishing emails. As mentioned in Section 3.3, the ham and spam corpora came from the SpamAssassin corpus, while the phishing corpus came from Jose Nazario. Each source took their own unique precautions to ensure the privacy of the emails by modifying some of the header information. For example, many of the messages in the phish corpus had their destination domain changed to *example.com* (see Figure 4-7). These modifications may be enough to provide a hint to the classifier and distinguish which corpus the message came from.

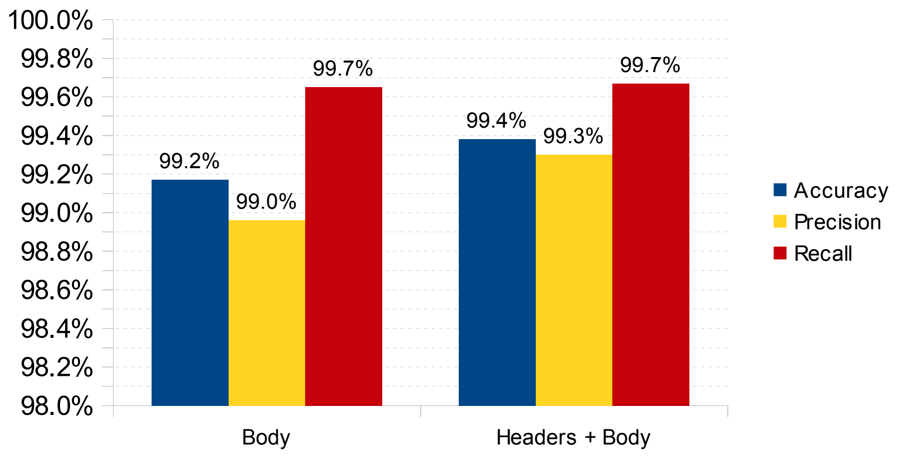

Figure 4-19: Performance Comparison on the Effect of Headers for Ham vs. Spam + Phish ($c = 500$)

A simple way check the effect of these inadvertent hints is to remove the headers from the feature set and focus entirely on the body of the message. Figure 4-19 compares the classification results with and without headers. From this figure it is possible to see the classifier still provides adequate performance without the headers, and any bias that may be introduced should be negligible.

4.7.2 Generative vs. Non-generative Classifiers

Another factor contributing to the high performance of the two-way classifier may be the use of a non-generative classifier. As explained in Section 2.5.5, most known email content filters use variations of Naive Bayes classifiers. Perhaps bayesian classifiers are prone to confusion when phish is introduced into the corpus.

To test this possible explanation, I ran the feature sets through Mallet's Naive Bayes classifier to see how it would perform with the same corpora and feature set both with and without phish. I used Mallet's built-in 10-way cross-validation feature which randomly divides the corpora into 10 equal parts and runs the 10-way cross-validation in the same way the previous testing was conducted: ham vs. spam, ham vs. spam + phish, and ham vs. spam vs. phish. Since Mallet also provides a Maximum Entropy classifier, both were run to provide validation on the performance of non-generative classifiers.

The results of the first test (ham vs. spam) immediately show the difference in classifier performance between generative (i.e. Naive Bayes) and non-generative (i.e. Maxent and SVM) classifiers, with Naive Bayes' performance being much lower than that of SVM or MaxEnt (see Figure 4-20). We also see that using the URL features improves non-generative performance while it degrades generative performance. This shows feature selection affects generative classifiers much more than non-generative ones.

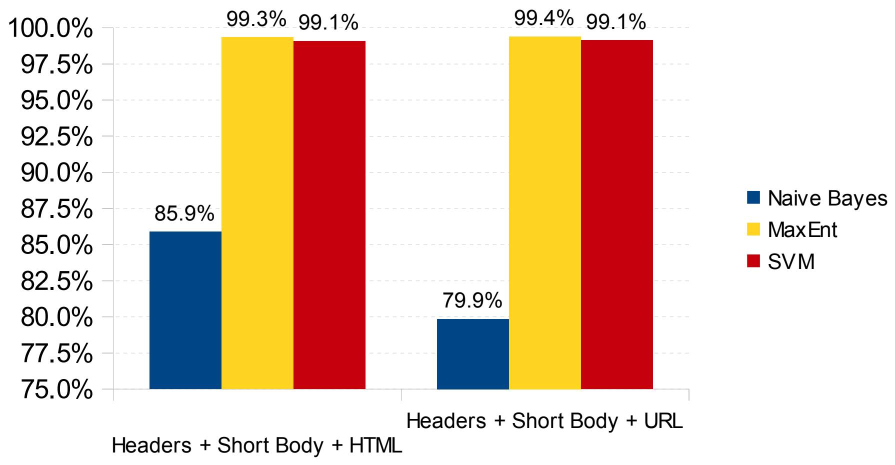

Figure 4-20: Accuracy Comparison Between Generative and Non-generative Models (Ham vs. Spam)

For the second test the phish corpus was added to the spam corpus and the test was run again (see Figure 4-21). Surprisingly, the performance of all three classifiers increased when phish was introduced to the corpora. This is due to the fact that each classifier was able to classify phish more accurately than ham or spam (see Appendix B for category-specific performance for SVM). These results show that with proper training, even Naive Bayes models are able to handle phishing emails.

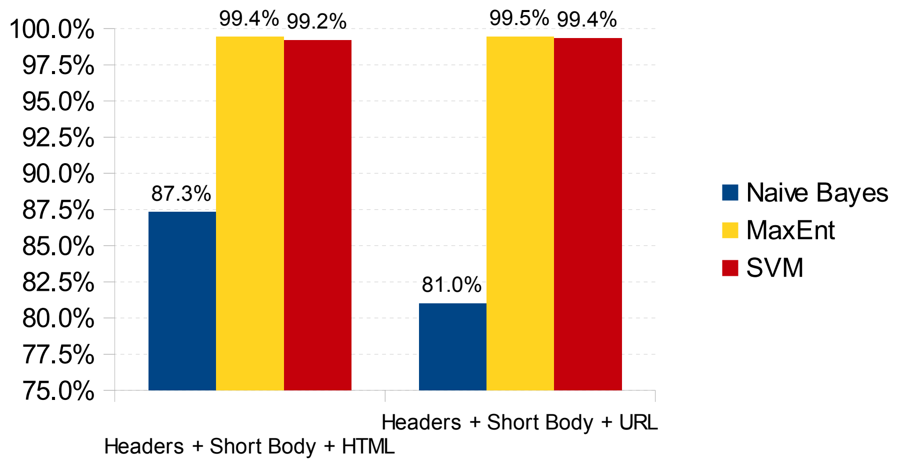

Figure 4-21: Accuracy Comparison Between Generative and Non-generative Models (Ham vs. Spam + Phish)

Finally, the three-way classifier offered similar performance to the two-way classifier when using MaxEnt, slightly worse performance using SVM, but significantly worse performance with Naive Bayes, proving once again that using three categories does not improve filtering performance (see Figure 4-22).

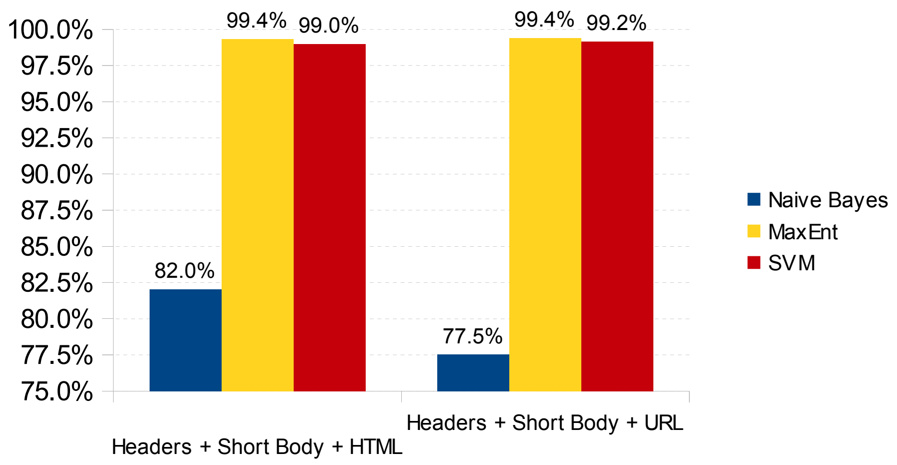

Figure 4-22: Accuracy Comparison Between Generative and Non-generative Models (Ham vs. Spam vs. Phish)

4.7.3 Feature Selection and Reduction

As was evident during the previous tests, feature selection appears to have a large impact on the performance of a Naive Bayes classifier. In order to improve performance with Naive Bayes, several sources suggest reducing the number of features. For example, Graham states that "another effect of a larger vocabulary [feature space] is that when you look at an incoming mail you find more interesting tokens [features], meaning those with probabilities far from .5. I use the 15 most interesting to decide if mail is spam" (Graham, 2003). Houser does not limit his filter to 15 features: ""Interesting tokens" were tokens in the message with a spam statistic "greater than 0.95"and "less than 0.05"' (Houser, n.d.). These types of reductions make sense, especially in probabilistic classifiers like Graham's and Houser's (variation of Naive Bayes) since they allow the filter to focus on the most telling features ("spamminess") that are most helpful to the classification problem. For example, when considering word-based features, ignoring common words such as "the," , "a," "an," "for,", "to," etc. allows the classifier to focus on more important spam features such as “buy,” “Viagra,” and “now!!!” However, the “interesting features” that make spam so obvious to the classifier are mostly absent from phish. Phishing attacks don’t try to get you to “buy Viagra now!!!” Thus, feature reduction deprives classifiers of the phish-specific features, causing a drop in classification performance.

Figure 4-23 Explores the performance variability as a product of feature set. By taking the accuracies of all common feature sets for the three classifiers from Appendix C, we can see Naive Bayes has a tremendous amount of variance in performance compared to non-generate classifiers. Thus we see that variations in features, whether from the source or the feature extractor, disrupt Naive Bayes classifiers but have little effect on non-generative classifiers.

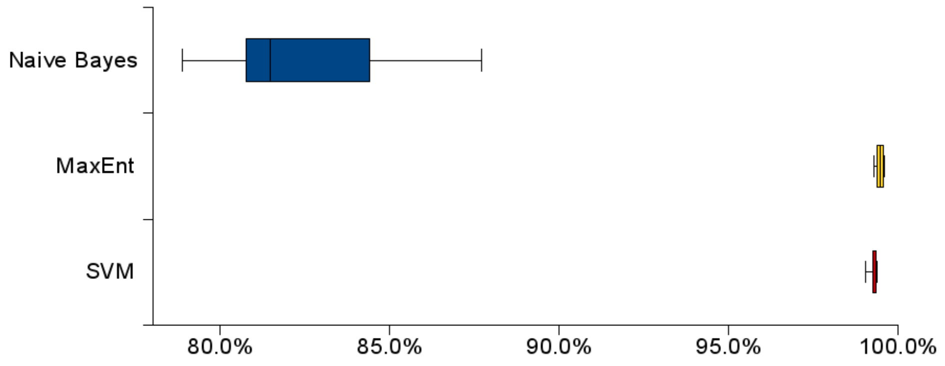

Figure 4-23: Performance Variance for Naive Bayes, Maximum Entropy and SVM Using Different Feature Sets

4.8 Validating the Results

To validate the results and test the validity of the assumptions in Section 4.7, the experiments were performed again on new corpora. If the results and assumptions are valid the results should be reproducible across different types of data.

The corpora chosen for validation are the first 10,000 entries of the 2005 TREC Public Spam Corpus for ham and spam, and sets 0, 1 (20051114) and 2 of Jose Nazario's phish corpus. These corpora provide equivalent volume of emails to the original corpora, although the number of messages in the ham and spam corpora are reversed as illustrated in Table 4-7. The high number of unwanted messages should also provide a more real-world testing scenario.

Table 4-7: Number of Messages in Corpora

Corpus	Number of Messages	Wanted/Unwanted Totals
Ham	2,126	2,126
Phish	7,874	10,154
Spam	2,280	

The data was run through the Short Body + HTML + URL feature set. This feature set is nearly identical to one of the best feature sets except that no features were extracted from any headers. This was to avoid any bias that may be introduced through the headers and focus purely on content.

The results confirm all the findings (see Appendix B). First, there was no degradation of performance when phishing is introduced. Second, blind phish classification results lean towards spam (95% of phish emails were blindly classified as spam in this case). Third, using a three-way classifier yielded nearly identical performance to the two-way classifier (see Figure 4-24).

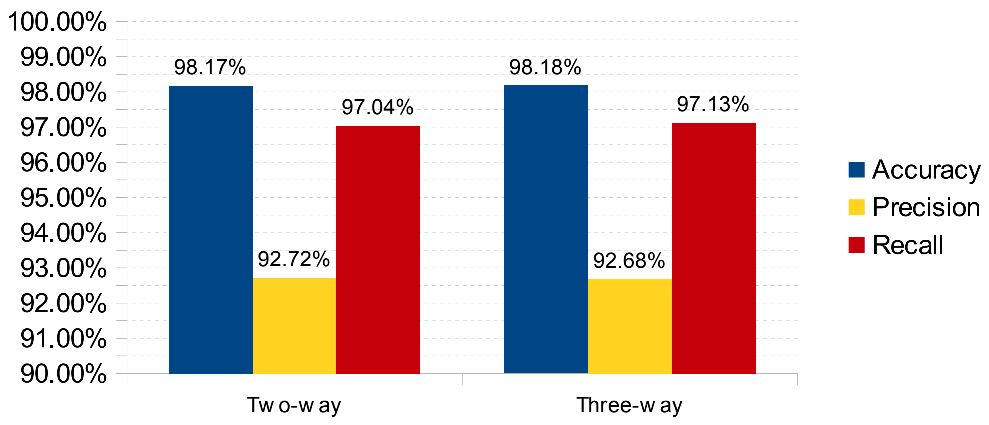

Figure 4-24: Performance of Two-way vs. Three-way Classifiers with the 2005 TREC Corpus (Headers + Short Body + URL, $c = 500$)

Testing these results for statistical significance also leads to the same conclusion: the difference in performance is *statistically insignificant* (p-value: 0.9805). This confirms the finding that there is no performance to be gained by using three-way classifiers (see Appendix D). The performance can be easily compared in Figure 4-25.

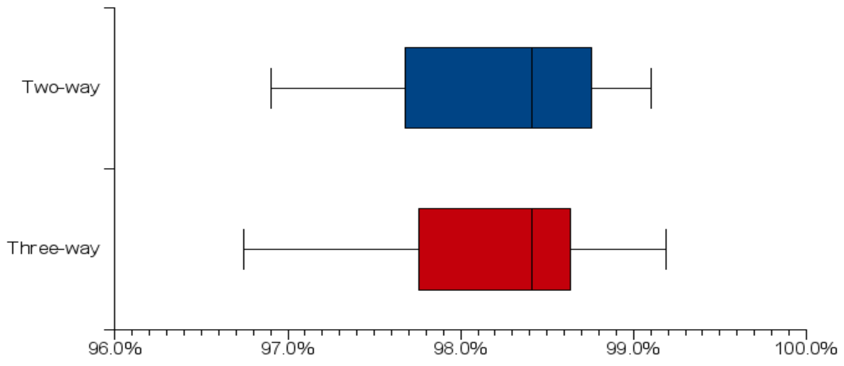

Figure 4-25: Comparison of Mean and Variance of the Accuracy of Two-way vs. Three-way Classifier for the 2005 TREC Corpus

5 CONCLUSIONS AND RECOMMENDATIONS

5.1 Conclusions

The purpose of the research for this thesis was to see if a spam filter's performance drop caused by phishing attacks could be reduced by creating a third classification category for phish. To test this hypothesis I gathered public email corpora with legitimate email (ham), spam and phishing emails and built a text classifier using a Support Vector Machine (SVM) to test how well it could classify the corpora in a traditional two-class configuration. Then, I classified the corpora using my proposed three-class configuration and compared the results.

The results and their analysis strongly suggest there is no statistically significant change in performance by using a three-way classifier. The difference in mean performance was 0.12% with a *p*-value of 0.48 (see Section 4.6). Additionally, the three-way classifier took three times more computing time than its two-way counterpart.

Part of the reason why the three-way classifier did not have a statistically significant improvement in performance was because the traditional two-way classifier performed admirably when phishing emails were introduced (see Section 4.5). It's performance was surprising since it contradicts the literature which suggests phishing emails cause a drop in classifier performance.

I further explored the apparent discrepancy between the literature and my results by running the same data through the same tests with a Naive Bayes classifier. Two things were very apparent. First, Naive Bayes had substantially lower performance on the same feature set than SVM (see Section 4.7.2). Second, the variance in Naive Bayes' classification performance was so large it proved variations in the training features, whether they come from the source data or the feature extraction process, can significantly disrupt performance (see Section 4.7.3).

In addition to running the data through a Naive Bayes classifier, I also ran the data through a Maximum Entropy classifier. It's performance was on par with SVM and did not suffer from the variability of Naive Bayes (see Sections 4.7.2 and 4.7.3), indicating that these non-generative classifiers are better suited to solve the spam and phish filtering problem.

Finally, to validate the findings, I ran all tests using a different corpora to ensure the results remained consistent. These new tests confirmed all findings (see Section 4.8).

5.2 Recommendations

Based on the results I recommend four things to improve spam and phish filtering to email administrators.

First recommendation: move away from Naive Bayes or "Bayesian" classifiers to a non-generative classifier such as Maximum Entropy or Support Vector Machines. The data shows both Maximum Entropy and SVM classifiers outperform Naive Bayes classifiers when using generic features (see Appendix C). This will provide administrators with three benefits:

1. Better performance both in terms of accuracy and precision.
2. Less variability in performance.
3. Significantly less time spent managing features and tuning performance.

Second recommendation: move away from word-based features to character-based features. As explained in Section 3.4 and demonstrated in section 4.2, character-based features provide much more robust features which deal better with obfuscation and extend effortlessly to foreign languages. Character-based features need to be paired with non-generative classifiers, as they do not offer good performance with Naive Bayes classifiers.

Third: ensure a proper and representative amount of phishing emails are included in the training corpus. The data has shown that adding an adequate amount of representative phishing emails to the training corpus will allow the classifier to achieve excellent results over “blind” phish classification (see and compare results in Sections 4.4 and 4.5). What constitutes an adequate amount and representative messages will depend on the type and volume of email handled by a particular domain.

Forth, training data should be divided into two categories: wanted and unwanted. The results suggest there is nothing to be gained by separating the unwanted category into phish and spam (or even other categories). If you are reading thesis trying to find a way to achieve better phish-filtering performance, follow the first three recommendations and do not create a third category.

5.3 Future Work

Since the corpora is rather old and limited in terms of the volume of global email, the results should be confirmed on newer, more representative and diverse corpora.

One of the reasons why Naive Bayes is still heavily used on the Internet is that there is no simple, freely available tool which provides non-generative spam filtering. I believe that if a simple tool that was easy to deploy in multiple environments was available, the Internet would move away from bayesian filters. The feature extractor in Appendix B provides a good foundation for such a tool.

Even though maximizing the performance of the filter was not a specific goal of this thesis, it became clear that more research should be done into obtaining better filter performance by improving feature extraction. As suggested by Kanaris, using n-grams of size 4, 5 and 6 offer better classification performance. Additionally, other features are possible candidates. For example, a new feature could create a character distribution on attachment filenames. Legitimate attachments appear have normal names, while spam and phishing emails usually don’t. Another feature could look at the *Received* headers. These are often forged by spammers and may carry a spammer’s signature.

Finally, the open source/academic communities need to create a holistic filter at the SMTP level. If content filter were combined with virus scanning, SPF/Sender ID, tarpitting, and some of the other techniques of Section 2.5, it should be possible to create an effective spam filtering solution that reduces security threats, bandwidth, storage, processing and the amount of unwanted email delivered to recipients.

Significant research should still be done in the spam filtering arena. In particular, we need to find simple yet effective solutions to the problem. The literature is replete with many approaches, few of which have provided applicable solutions and fewer have made it to the open source community.

REFERENCES

2005 TREC Public Spam Corpus. 2005. http://plg.uwaterloo.ca/~gvcormac/treccorpus/ (accessed February 27, 2012).

Allman, E., J. Callas, M. Delany, M. Libbey, J. Fenton, and M. Thomas. “RFC 4871: DomainKeys Identfied Mail (DKIM) Signatures”. Internet Engineering Task Force, May 2007. http://tools.ietf.org/html/rfc4871.

Androutsopoulos, I. Ling-Spam datasets - Csmining Group, 2000. http://csmining.org/index.php/ling-spam-datasets.html. (accessed February 6, 2012).

Apache Foundation. 2004. ASF Position Regarding Sender ID. September 2, 2004. http://www.apache.org/foundation/docs/sender-id-position.html. (accessed February 6, 2012).

APWG. 2007. Phishing Activities Trends Report, 1st Half / 2011. Anti-Phishing Working Group, December 23, 2011. http://www.antiphishing.org/reports/apwg_trends_report_h1_2011.pdf (accessed January 30, 2012).

Barracuda Networks. Barracuda Spam & Virus Firewall - Comprehensive Spam and Virus Protection, n.d. http://www.barracudanetworks.com/ns/products/spam_overview.php. (accessed February 6, 2012).

Bergholz, A., G. Paaß, F. Reichartz, S. Strobel, and J. Chang. 2008. “Improved Phishing Detection using Model-Based Features”. In Conference on Email and Anti-Spam (CEAS).

Caruana, R., and A. Niculescu-Mizil. 2006. An Empirical Comparison of Supervised Learning Algorithms. In Proceedings of the 23rd International Conference on Machine Learning. Pittsburgh, PA.

Chhabra, S. 2005. Fighting Spam, Phishing and Email Fraud. Riverside, CA: University of California Riverside. http://www.cs.ucr.edu/~schhabra/ack.pdf.

Crammer K. and Y. Singer. 2001. On the Algorithmic Implementation of Multi-class SVMs. Journal of Machine Learning Research. December 2001. http://www.cis.uab.edu/zhang/Spam-mining-papers/ On.the.Algorithmic.Implementation.of.Multiclass.Kernel.based.Vector.machines.pdf.

Crocker, D. 1982. RFC 822: Standard for the Format of ARPA Internet Text Messages. Internet Engineering Task Force, August 13, 1982. http://tools.ietf.org/html/rfc822.

Dacier, M. 2009. Technical Perspective: They Do Click, Don't They? Communications of the ACM 52, no. 9 (September 2009).

DMARC.org. Domain-based Message Authentication, Reporting and Conformance, n.d. http://www.dmarc.org/. (accessed February 6, 2012).

Eggendorfer, T. 2007. Reducing spam to 20% of its original value with a SMTP tar pit simulator. In MIT Spam Conference. Cambridge, MA, 2007.

FBI. Common Fraud Schemes, n.d. http://www.fbi.gov/scams-safety/fraud. (accessed February 6, 2012).

FBI. 2009. Spear Phishing. April 1, 2009. http://www.fbi.gov/news/stories/2009/april/spearphishing_040109. (accessed February 6, 2012).

Fette, I., N. Sadeh, and A. Tomasic. 2007. Learning to Detect Phishing Emails. Proceedings of the International World Wide Web Conference (WWW). Banff, Alberta. http://www.cs.cmu.edu/~tomasic/doc/2007/FetteSadehTomasicWWW2007.pdf.

Google. Gmail – Google Apps, n.d. http://www.google.com/apps/intl/en/business/gmail.html. (accessed February 6, 2012).

Goodman, J., G. Cormack, and D. Heckerman. Spam and the Ongoing Battle for the Inbox. Communications of the ACM 50, no. 2 (February 2007): 25-33.

Graham, P. 2003. Better Bayesian Filtering, January 2003. http://www.paulgraham.com/better.html (accessed November 17, 2011).

Hanna, J. ASSP, n.d. http://assp.sourceforge.net/.

Houser, S. Free Unix Shell Statistical Spam Filter and Whitelist, n.d. http://sofbot.com/article/Statistical_spam_filter.html (accessed January 2, 2012).

Hutzler, C., D. Crocker, P. Resnick, E. Allman, and T. Finch. RFC 5068: Email Submission Operations: Access and Accountability Requirements. Internet Engineering Task Force, November 2007. http://tools.ietf.org/html/rfc5068.

Jacob, P. 2003. The SPAM Problem: Moving Beyond RBLs. January 3, 2003. http://www.whirlycott.com/phil/antispam/rbl-bad/rbl-bad.html (accessed February 6, 2012).

Joachims, T. 2002. Learning to Classify Text Using Support Vector Machines: Methods Theory and Algorithms. Norwell, MA: Kluwer Academic Publishers, 2002.

Kavi Help Center. How Email Really Works. http://www.niso.org/khelp/kmlm/user_help/html/how_email_works.html (accessed January 24, 2012).

Kanich, C., C. Kreibich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. 2009. Spamalytics: An Empirical Analysis of Spam Marketing Conversion. Communications of the ACM 52, no. 9 (September 2009): 99-107.

Kanaris, I., K. Kanaris, and E. Stamatatos. 2006. Spam Detection Using Character N-Grams. 3955:95-104. Greece: Lecture Notes in Artificial Intelligence.

Klensin, J. RFC 2821: Simple Mail Transfer Protocol. Internet Engineering Task Force, April 2001. http://tools.ietf.org/html/rfc2821.

Klensin, J. RFC 5321: Simple Mail Transfer Protocol. Internet Engineering Task Force, October 2008. http://tools.ietf.org/html/rfc5321.

Leyden, J. 2011. IPv6 intro creates spam-filtering nightmare. The Register, March 8, 2011. http://www.theregister.co.uk/2011/03/08/ipv6_spam_filtering_headache/ (accessed March 8, 2011).

Leyden, J. 2012. Kelihos botnet BACK FROM THE DEAD. The Register. London, UK, February 2, 2012. http://www.theregister.co.uk/2012/02/02/kelihos_botnet_returns/ (accessed February 5, 2012).

Lindberg, G. 1999. RFC 2505: Anti-Spam Recommendations for SMTP MTAs. Internet Engineering Task Force, February 1999. http://tools.ietf.org/html/rfc2505.

Lyon, J., and M. Wong. RFC 4406: Sender ID: Authenticating E-Mail. Internet Engineering Task Force, April 2006. http://tools.ietf.org/html/rfc4406.

M86 Security. 2008. Srizbi now leads the spam pack. February 29, 2008. http://www.m86security.com/labs/traceitem.asp?article=567 (accessed April 6, 2011).

MAAWG. 2011. Email Metrics Program: The Network Operators' Perspective, Report #15 – First, Second and Third Quarter 2011. Messaging Anti-Abuse Working Group. http://www.maawg.org/sites/maawg/files/news/MAAWG_2011_Q1Q2Q3_Metrics_Report_15.pdf.

Manning, C. and H. Schütze. 2000. Foundations of Statistical Natural Language Processing. The MIT Press.

MailRoute. MailRoute: Email Protection - Spam and Virus Filtering Services, n.d. http://mailroute.info/services.html#spam (accessed February 8, 2011).

McCallum, A. 2002. MALLET: A Machine Learning for Language Toolkit. http://mallet.cs.umass.edu.

Nasario, J.. Monkey.org's Phishing Email Corpus, n.d. http://www.monkey.org/~jose/wiki/doku.php?id=phishingcorpus (accessed February 22, 2011).

NLTK Documentation. n.d. Learning to Classify Text http://nltk.googlecode.com/svn/trunk/doc/book/ch06.html (accessed February 6, 2012).

Ollmann, G. The Phishing Guide (Part 1). http://www.technicalinfo.net/papers/Phishing.html (accessed January 24, 2012).

Partridge, C. 1986. RFC 974: Mail Routing and the Domain System. Internet Engineering Task Force, Jaunary 1986. http://tools.ietf.org/html/rfc974.

Postel, J. 1982. RFC 821: Simple Mail Transfer Protocol. Internet Engineering Task Force, August 1982. http://tools.ietf.org/html/rfc821.

Ramsey, F. and D. Schafer. 2002. The Statistical Sleuth: A Course in Methods of Data Analysis. Second ed. Belmont, CA: Brooks/Cole, 2002.

Reynolds, J. K. 1984. RFC 918: Post Office Protocol. Internet Engineering Task Force, October 1984. http://tools.ietf.org/html/rfc918.

Ringger, E. 2005. Proceedings of the Workshop. In ACL 2005 Workshop on Feature Engineering for Machine Learning in Natural Language Processing. Ann Arbor, Michigan, 2005.

Ringger, E. 2012. Personal email to author. Feburary 28, 2012.

Seabrook, A.. At 30, Spam Going Nowhere Soon : NPR, May 2008. http://www.npr.org/templates/story/story.php?storyId=90160617 (accessed April 6, 2011).

Securelist, n.d. Types of Spam. http://www.securelist.com/en/threats/spam?chapter=88 (accessed November 23, 2011).

SpamAssassin Public Corpus, n.d. http://spamassassin.apache.org/publiccorpus/ (accessed February 6, 2012).

SpamAssassin. Tests Performed. The Apache SpamAssassin Project, n.d. http://spamassassin.apache.org/tests.html (accessed February 19, 2011).

SpamBayes. n.d. SpamBayes: Background Reading. SpamBayes Website. http://spambayes.sourceforge.net/background.html (accessed February 6, 2012).

SpamCop FAQ. Why are auto responders bad? n.d. http://www.spamcop.net/fom-serve/cache/329.html (accessed February 6, 2012).

Spamhous Project. About Spamhouse. http://www.spamhaus.org/organization/index.lasso (accessed January 31, 2012).

Symantec. Spam Filter Service - Email Spam Filtering - Symantec.Cloud, n.d. http://www.symanteccloud.com/products/email/anti_spam.aspx (accessed February 6, 2012).

Templeton, B. n.d. Reaction to the DEC Spam of 1978. http://www.templetons.com/brad/spamreact.html (accessed April 6, 2011).

Templeton, Brad. n.d. Proper principles for Challenge/Response anti-spam systems. http://www.templetons.com/brad/spam/challengeresponse.html (accessed April 6, 2011).

Treviño, A. and J. Ekstrom. 2007a. Email sender authentication: advantages and shortcomings. Virus Bulletin, August 1, 2007.

Treviño, A. and J. Ekstrom. 2007b. Spam Filtering Through Header Relay Detection. In MIT Spam Conference. Cambridge, MA, 2007.

Websense. Websense Advanced Classification Engine, n.d. http://www.websense.com/content/websense-advanced-classification-engine.aspx (accessed November 22, 2011).

Wong, M., and W. Schlitt. 2006. RFC 4408: Sender Policy Framework (SPF) for Authorizing Use Domains in E-Mail, Version 1. Internet Engineering Task Force, April 2006. http://tools.ietf.org/html/rfc4408.

Yamai, N., K. Okayama, T. Seike, K. Kawano, M. Nakamura, and S. Maruyama. 2008. An Anti-Spam Method with SMTP Session Abort. MIT Spam Conference 2008. http://projects.csail.mit.edu/spamconf/SC2008/Yamai_SMTP-session-abort/yamai_SMTP_session_abort.pdf.

Youngman, N. 2004. Fine-tuning SpamAssassin LG #105. Linux Gazette, August 2004. http://linuxgazette.net/105/youngman.html. (accessed February 6, 2012).

Zdziarski, J. 2005. Ending Spam: Bayesian Content Filtering and the Art of Statistical Language Classification. No Starch Press.

Zdziarski, J., W. Yang, and P. Judge. 2006. Approaches to Phishing Identification Using Match and Probabilistic Digital Fingerprinting Techniques. In MIT Spam Conference. Cambridge, MA, 2006.

Zhang L., J. Zhu, T. Yao. 2004. An evaluation of statistical spam filtering techniques. ACM Transactions on Asian Language Information Processing (TALIP) 3, no. 4 (December 2004).

APPENDIX A: FEATURE SETS

The following table contains a list of the different feature sets producible by the feature extractor. Each set contains the feature set name, the way it was referenced in the text, and the specific features it includes. (References to *short body* means only the first and last 1,024 characters of a body part were used.)

Feature Set Descriptions		
Feature Set	**Text Reference**	**Features**
unigram	Unigram	• Unigram distribution of message body
bigram	Unigram + Bigram	• Unigram and bigram distributions of message body
trigram	Unigram + Bigram + Trigram / Body	• Unigram, bigram and trigram distributions of message body
headers_only	Headers Only	• Unigram, bigram and trigram distributions of select headers
headers_body	Headers + Body	• Unigram, bigram and trigram distributions of: ○ Select headers ○ Message body
headers_body_count	Headers + Body + Counts	• Unigram, bigram and trigram distributions of: ○ Select headers ○ Message body • Multi-part counts
headers_body_split	Headers + Body + HTML	• Unigram, bigram and trigram distributions of: ○ Select headers ○ Plain text body ○ HTML body
headers_body_split_ strip	Headers + Body + HTML Text	• Unigram, bigram and trigram distributions of: ○ Select headers ○ Plain text body ○ HTML body ○ HTML text (no HTML tags)

Feature Set Descriptions		
Feature Set	**Text Reference**	**Features**
short_body	Short Body	• Unigram, bigram and trigram distributions of short message body
headers_short_body	Headers + Short Body	• Unigram, bigram and trigram distributions of: ○ Select headers ○ Short message body
headers_short_ body_strip	Headers + Short Body + HTML Text	• Unigram, bigram and trigram distributions of: ○ Select headers ○ Short plain text body and HTML text
headers_short_ body_split_strip	Headers + Short Body + HTML Text	• Unigram, bigram and trigram distributions of: ○ Short plain text body ○ Short HTML body ○ Short HTML text (no HTML tags)
full	Headers + Body + HTML Text + URL	• Unigram, bigram and trigram distributions of: ○ Select headers ○ Plain text body ○ HTML body ○ HTML text (no HTML tags) • Multi-part counts • URL analysis
full_short	Headers + Short Body + HTML Text + URL	• Unigram, bigram and trigram distributions of: ○ Select headers ○ Short plain text body ○ Short HTML body ○ Short HTML text (no HTML tags) • Multi-part counts • URL analysis
full_short_no_split	Headers + Short Body + HTML + URL	• Unigram, bigram and trigram distributions of: ○ Select headers ○ Short plain text body, short HTML body and short HTML text (no HTML tags) • Multi-part counts • URL analysis
full_short_no_split_ no_count	Headers + Short Body + HTML + URL	• Unigram, bigram and trigram distributions of: ○ Select headers ○ Short plain text body, short HTML body and short HTML text (no HTML tags) • URL analysis

Feature Set Descriptions		
Feature Set	**Text Reference**	**Features**
full_short_no_split_ no_headers	Short Body + HTML + URL	• Unigram, bigram and trigram distributions of: ○ Short plain text body, short HTML body and short HTML text (no HTML tags) • Multi-part counts • URL analysis

APPENDIX B: RAW CLASSIFIER RESULTS

The following are the full results of the classification tests for Section 4.2.

Ham vs. Spam Results (*c* = 100)							
Feature Set	**Raw Results (Confusion Matrix)**				**Performance**		**Running Time**
		Ham	**Spam**	**Phish**			
unigram	Ham	6794	160	–	Accuracy	93.37%	0:00:59
	Spam	462	1972	–	Precision	93.63%	
	Phish	–	–	–	Recall	97.70%	
bigram	Ham	6915	39	–	Accuracy	98.45%	0:02:53
	Spam	106	2292	–	Precision	98.49%	
	Phish	–	–	–	Recall	99.44%	
trigram	Ham	6930	24	–	Accuracy	98.90%	0:11:06
	Spam	79	2319	–	Precision	98.87%	
	Phish	–	–	–	Recall	99.65%	
headers_only	Ham	6774	180	–	Accuracy	95.61%	0:00:58
	Spam	231	2167	–	Precision	96.70%	
	Phish	–	–	–	Recall	97.41%	
headers_body	Ham	6931	23	–	Accuracy	99.24%	0:16:39
	Spam	48	2341	–	Precision	99.31%	
	Phish	–	–	–	Recall	99.67%	
headers_body_ count	Ham	6931	23	–	Accuracy	99.24%	0:16:33
	Spam	48	2350	–	Precision	99.31%	
	Phish	–	–	–	Recall	99.67%	
headers_body_ split	Ham	6931	23	–	Accuracy	99.17%	0:21:30
	Spam	55	2343	–	Precision	99.21%	
	Phish	–	–	–	Recall	99.67%	
headers_body_ split_strip	Ham	6931	23	–	Accuracy	99.14%	0:29:46
	Spam	57	2341	–	Precision	99.18%	

Ham vs. Spam Results (*c* = 100)							
Feature Set	**Raw Results (Confusion Matrix)**				**Performance**		**Running Time**
		Ham	**Spam**	**Phish**			
	Phish	–	–	–	Recall	99.67%	
short_body	Ham	6930	24	–	Accuracy	98.85%	0:07:53
	Spam	84	2314	–	Precision	98.80%	
	Phish	–	–	–	Recall	99.65%	
headers_short_body	Ham	6931	23	–	Accuracy	99.12%	0:10:39
	Spam	59	2339	–	Precision	99.16%	
	Phish	–	–	–	Recall	99.67%	
headers_short_body_strip	Ham	6931	23	–	Accuracy	99.12%	0:10:40
	Spam	59	2339	–	Precision	99.16%	
	Phish	–	–	–	Recall	99.67%	
headers_short_body_split_strip	Ham	6927	27	–	Accuracy	99.07%	0:11:59
	Spam	60	2338	–	Precision	99.14%	
	Phish	–	–	–	Recall	99.61%	
full	Ham	6931	23	–	Accuracy	99.16%	0:30:47
	Spam	56	2342	–	Precision	99.20%	
	Phish	–	–	–	Recall	99.67%	
full_short	Ham	6928	26	–	Accuracy	99.04%	0:11:49
	Spam	64	2334	–	Precision	99.08%	
	Phish	–	–	–	Recall	99.63%	
full_short_no_split	Ham	6930	24	–	Accuracy	99.13%	0:10:33
	Spam	57	2341	–	Precision	99.18%	
	Phish	–	–	–	Recall	99.65%	
full_short_no_split_no_count	Ham	6932	22	–	Accuracy	99.12%	0:10:33
	Spam	60	2338	–	Precision	99.14%	
	Phish	–	–	–	Recall	99.68%	

The following are the full results of the classification tests in Section 4.3 for the purposes of finding the optimal value of *c*.

Ham vs. Spam Results								
Feature Set	***c***	**Raw Results (Confusion Matrix)**				**Performance**		**Running Time**
			Ham	**Spam**	**Phish**			
full	0.1	Ham	6861	93	–	Accuracy	88.18%	0:07:00
		Spam	1012	1386	–	Precision	87.15%	
		Phish	–	–	–	Recall	98.66%	
full	1	Ham	6892	62	–	Accuracy	94.93%	0:11:04
		Spam	412	1986	–	Precision	94.36%	
		Phish	–	–	–	Recall	99.11%	
full	10	Ham	6923	31	–	Accuracy	98.29%	0:16:47
		Spam	129	2269	–	Precision	98.17%	
		Phish	–	–	–	Recall	99.55%	
full	100	Ham	6931	23	–	Accuracy	99.16%	0:30:49
		Spam	56	2342	–	Precision	99.20%	
		Phish	–	–	–	Recall	99.67%	
full	500	Ham	6929	25	–	Accuracy	99.17%	0:54:10
		Spam	53	2345	–	Precision	99.24%	
		Phish	–	–	–	Recall	99.64%	
full	1000	Ham	6929	25	–	Accuracy	99.17%	0:54:15
		Spam	53	2345	–	Precision	99.24%	
		Phish	–	–	–	Recall	99.64%	
full	5000	Ham	6929	25	–	Accuracy	99.17%	0:54:13
		Spam	53	2345	–	Precision	99.24%	
		Phish	–	–	–	Recall	99.64%	
full	10000	Ham	6929	25	–	Accuracy	99.17%	0:54:19
		Spam	53	2345	–	Precision	99.24%	
		Phish	–	–	–	Recall	99.64%	
full_short	0.1	Ham	6633	311	–	Accuracy	84.28%	0:03:30
		Spam	1158	1240	–	Precision	85.14%	

Ham vs. Spam Results								
Feature Set	*c*	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
		Phish	–	–	–	Recall	95.52%	
full_short	1	Ham	6847	107	–	Accuracy	91.91%	0:04:55
		Spam	650	1748	–	Precision	91.33%	
		Phish	–	–	–	Recall	98.46%	
full_short	10	Ham	6884	70	–	Accuracy	97.81%	0:07:54
		Spam	135	2263	–	Precision	98.08%	
		Phish	–	–	–	Recall	98.99%	
full_short	100	Ham	6928	26	–	Accuracy	99.04%	0:11:46
		Spam	64	2334	–	Precision	99.08%	
		Phish	–	–	–	Recall	99.63%	
full_short	500	Ham	6932	22	–	Accuracy	99.16%	0:20:53
		Spam	57	2341	–	Precision	99.18%	
		Phish	–	–	–	Recall	99.68%	
full_short	1000	Ham	6931	23	–	Accuracy	99.16%	0:21:10
		Spam	56	2342	–	Precision	99.20%	
		Phish	–	–	–	Recall	99.67%	
full_short	5000	Ham	6931	23	–	Accuracy	99.16%	0:21:09
		Spam	56	2342	–	Precision	99.20%	
		Phish	–	–	–	Recall	99.67%	
full_short	10000	Ham	6931	23	–	Accuracy	99.16%	0:21:10
		Spam	56	2342	–	Precision	99.20%	
		Phish	–	–	–	Recall	99.67%	
full_short_ no_split	0.1	Ham	6696	258	–	Accuracy	82.68%	0:03:18
		Spam	1362	1036	–	Precision	83.10%	
		Phish	–	–	–	Recall	96.29%	
full_short_ no_split	1	Ham	6862	92	–	Accuracy	92.62%	0:04:55
		Spam	598	1800	–	Precision	91.98%	
		Phish	–	–	–	Recall	98.68%	

Ham vs. Spam Results								
Feature Set	c	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
full_short_ no_split	10	Ham	6897	57	–	Accuracy	98.08%	0:07:14
		Spam	123	2275	–	Precision	98.25%	
		Phish	–	–	–	Recall	99.18%	
full_short_ no_split	100	Ham	6930	24	–	Accuracy	99.13%	0:10:31
		Spam	57	2341	–	Precision	99.18%	
		Phish	–	–	–	Recall	99.65%	
full_short_ no_split	500	Ham	6934	20	–	Accuracy	99.21%	0:18:09
		Spam	54	2344	–	Precision	99.23%	
		Phish	–	–	–	Recall	99.71%	
full_short_ no_split	1000	Ham	6936	18	–	Accuracy	99.22%	0:18:43
		Spam	55	2343	–	Precision	99.21%	
		Phish	–	–	–	Recall	99.74%	
full_short_ no_split	5000	Ham	6936	18	–	Accuracy	99.22%	0:18:48
		Spam	55	2343	–	Precision	99.21%	
		Phish	–	–	–	Recall	99.74%	
full_short_ no_split	10000	Ham	6936	18	–	Accuracy	99.22%	0:18:45
		Spam	55	2343	–	Precision	99.21%	
		Phish	–	–	–	Recall	99.74%	
full_short_ no_split_ no_count	0.1	Ham	6705	249	–	Accuracy	82.80%	0:03:18
		Spam	1360	1038	–	Precision	83.14%	
		Phish	–	–	–	Recall	96.42%	
full_short_ no_split_ no_count	1	Ham	6857	97	–	Accuracy	92.60%	0:04:59
		Spam	595	1803	–	Precision	92.02%	
		Phish	–	–	–	Recall	98.61%	
full_short_ no_split_ no_count	10	Ham	6895	59	–	Accuracy	98.09%	0:07:13
		Spam	120	2278	–	Precision	98.29%	
		Phish	–	–	–	Recall	99.15%	
full_short_	100	Ham	6932	22	–	Accuracy	99.13%	0:10:30

Ham vs. Spam Results								
Feature Set	c	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
no_split_ no_count		Spam	60	2388	–	Precision	99.14%	
		Phish	–	–	–	Recall	99.68%	
full_short_ no_split_ no_count	500	Ham	6935	19	–	Accuracy	99.19%	0:18:10
		Spam	57	2341	–	Precision	99.18%	
		Phish	–	–	–	Recall	99.73%	
full_short_ no_split_ no_count	1000	Ham	6935	19	–	Accuracy	99.22%	0:18:57
		Spam	54	2344	–	Precision	99.23%	
		Phish	–	–	–	Recall	99.73%	
full_short_ no_split_ no_count	5000	Ham	6935	19	–	Accuracy	99.22%	0:18:59
		Spam	54	2344	–	Precision	99.23%	
		Phish	–	–	–	Recall	99.73%	
full_short_ no_split_ no_count	10000	Ham	6935	19	–	Accuracy	99.22%	0:18:59
		Spam	54	2344	–	Precision	99.23%	
		Phish	–	–	–	Recall	99.73%	

The following are the full results of the classification tests in Section 4.4.

Blind Phish Results								
Feature Set	c	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
headers_body_ split_strip	100	Phish	598	1677	–	Accuracy	73.71%	0:03:18
headers_body_ split_strip	500	Phish	426	1849	–	Accuracy	81.27%	0:06:52
headers_body_ split_strip	1000	Phish	426	1849	–	Accuracy	81.27%	0:06:53
headers_body_ split_strip	5000	Phish	426	1849	–	Accuracy	81.27%	0:06:49

Blind Phish Results								
Feature Set	c	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
headers_short_ body_strip	100	Phish	637	1638	–	Accuracy	72.00%	0:01:12
headers_short_ body_strip	500	Phish	604	1671	–	Accuracy	73.45%	0:01:57
headers_short_ body_strip	1000	Phish	591	1684	–	Accuracy	74.02%	0:02:07
headers_short_ body_strip	5000	Phish	591	1684	–	Accuracy	74.02%	0:02:07
headers_short_ body_split_strip	100	Phish	425	1850	–	Accuracy	81.32%	0:01:22
headers_short_ body_split_strip	500	Phish	266	2009	–	Accuracy	88.31%	0:02:11
headers_short_ body_split_strip	1000	Phish	182	2093	–	Accuracy	92.00%	0:02:14
headers_short_ body_split_strip	5000	Phish	182	2093	–	Accuracy	92.00%	0:02:14
full	100	Phish	495	1780	–	Accuracy	78.24%	0:03:31
full	500	Phish	567	1708	–	Accuracy	75.08%	0:05:18
full	1000	Phish	567	1708	–	Accuracy	75.08%	0:05:16
full	5000	Phish	567	1708	–	Accuracy	75.08%	0:05:14
full_short	100	Phish	362	1913	–	Accuracy	84.09%	0:01:26
full_short	500	Phish	148	2127	–	Accuracy	93.49%	0:02:21
full_short	1000	Phish	224	2051	–	Accuracy	90.15%	0:02:23
full_short	5000	Phish	224	2051	–	Accuracy	90.15%	0:02:20
full_short_ no_split	100	Phish	660	1615	–	Accuracy	70.99%	0:01:10
full_short_ no_split	500	Phish	578	1697	–	Accuracy	74.59%	0:02:04
full_short_ no_split	1000	Phish	597	1678	–	Accuracy	73.76%	0:02:17
full_short_ no_split	5000	Phish	597	1678	–	Accuracy	73.76%	0:02:18

Blind Phish Results								
Feature Set	*c*	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
full_short_no_ split_no_count	100	Phish	655	1620	–	Accuracy	71.21%	0:01:11
full_short_no_ split_no_count	500	Phish	503	1772	–	Accuracy	77.89%	0:01:55
full_short_no_ split_no_count	1000	Phish	619	1656	–	Accuracy	72.79%	0:02:01
full_short_no_ split_no_count	5000	Phish	619	1656	–	Accuracy	72.79%	0:02:02

The following are the full classification results for tests in Section 4.5.

Ham vs. Spam + Phish Results								
Feature Set	*c*	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
trigram (body)	100	Ham	6926	28	–	Accuracy	98.92%	0:16:25
		Spam	71	2327	–	Precision	98.60%	
		Phish	27	2248	–	Recall	99.60%	
trigram (body)	500	Ham	6930	24	–	Accuracy	99.17%	0:25:10
		Spam	48	2350	–	Precision	98.96%	
		Phish	25	2250	–	Recall	99.65%	
trigram (body)	1000	Ham	6932	22	–	Accuracy	99.16%	0:31:48
		Spam	50	2348	–	Precision	98.92%	
		Phish	26	2249	–	Recall	99.68%	
trigram (body)	5000	Ham	6931	23	–	Accuracy	99.16%	0:56:38
		Spam	47	2351	–	Precision	98.93%	
		Phish	28	2247	–	Recall	99.67%	
headers_body	100	Ham	6929	25	–	Accuracy	99.33%	0:22:42
		Spam	43	2355	–	Precision	99.24%	

Ham vs. Spam + Phish Results								
Feature Set	c	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
		Phish	10	2265	–	Recall	99.64%	
headers_body	500	Ham	6931	23	–	Accuracy	99.38%	0:39:10
		Spam	40	2358	–	Precision	99.30%	
		Phish	9	2266	–	Recall	99.67%	
headers_body	1000	Ham	6932	22	–	Accuracy	99.42%	0:43:06
		Spam	37	2361	–	Precision	99.36%	
		Phish	8	2267	–	Recall	99.68%	
headers_body	5000	Ham	6932	22	–	Accuracy	99.42%	0:43:18
		Spam	37	2361	–	Precision	99.36%	
		Phish	8	2267	–	Recall	99.68%	
headers_body_ split_strip	100	Ham	6924	30	–	Accuracy	99.22%	0:42:16
		Spam	51	2347	–	Precision	99.13%	
		Phish	10	2265	–	Recall	99.57%	
headers_body_ split_strip	500	Ham	6929	25	–	Accuracy	99.33%	1:19:02
		Spam	45	2353	–	Precision	99.24%	
		Phish	8	2267	–	Recall	99.64%	
headers_body_ split_strip	1000	Ham	6930	24	–	Accuracy	99.35%	1:22:01
		Spam	45	2353	–	Precision	99.26%	
		Phish	7	2268	–	Recall	99.65%	
headers_body_ split_strip	5000	Ham	6930	24	–	Accuracy	99.35%	1:22:02
		Spam	45	2353	–	Precision	99.26%	
		Phish	7	2268	–	Recall	99.65%	
short_body	100	Ham	6926	28	–	Accuracy	98.92%	0:10:37
		Spam	74	2324	–	Precision	98.60%	
		Phish	24	2251	–	Recall	99.60%	
short_body	500	Ham	6639	18	–	Accuracy	99.03%	0:15:59
		Spam	66	2332	–	Precision	98.63%	
		Phish	26	2249	–	Recall	99.73%	
short_body	1000	Ham	6936	18	–	Accuracy	99.17%	0:20:50

Ham vs. Spam + Phish Results								
Feature Set	***c***	**Raw Results (Confusion Matrix)**				**Performance**		**Running Time**
			Ham	**Spam**	**Phish**			
		Spam	56	2342	–	Precision	98.89%	
		Phish	22	2253	–	Recall	99.74%	
short_body	5000	Ham	6937	17	–	Accuracy	99.15%	0:38:04
		Spam	58	2340	–	Precision	98.83%	
		Phish	24	2251	–	Recall	99.76%	
headers_short_ body	100	Ham	6927	27	–	Accuracy	99.24%	0:14:18
		Spam	48	2350	–	Precision	99.13%	
		Phish	13	2262	–	Recall	99.61%	
headers_short_ body	500	Ham	6933	21	–	Accuracy	99.36%	0:23:28
		Spam	47	2351	–	Precision	99.24%	
		Phish	6	2269	–	Recall	99.70%	
headers_short_ body	1000	Ham	6932	22	–	Accuracy	99.32%	0:25:50
		Spam	50	2348	–	Precision	99.18%	
		Phish	7	2268	–	Recall	99.68%	
headers_short_ body	5000	Ham	6932	22	–	Accuracy	99.32%	0:25:43
		Spam	50	2348	–	Precision	99.18%	
		Phish	7	2268	–	Recall	99.68%	
headers_short_ body_strip	100	Ham	6927	27	–	Accuracy	99.24%	0:14:08
		Spam	48	2350	–	Precision	99.13%	
		Phish	13	2262	–	Recall	99.61%	
headers_short_ body_strip	500	Ham	6933	21	–	Accuracy	99.36%	0:23:02
		Spam	47	2351	–	Precision	99.24%	
		Phish	6	2269	–	Recall	99.70%	
headers_short_ body_strip	1000	Ham	6932	22	–	Accuracy	99.32%	0:25:13
		Spam	50	2348	–	Precision	99.18%	
		Phish	7	2268	–	Recall	99.68%	
headers_short_ body_strip	5000	Ham	6932	22	–	Accuracy	99.32%	0:25:14
		Spam	50	2348	–	Precision	99.18%	
		Phish	7	2268	–	Recall	99.68%	

Ham vs. Spam + Phish Results								
Feature Set	***c***	**Raw Results (Confusion Matrix)**				**Performance**		**Running Time**
			Ham	**Spam**	**Phish**			
headers_short_ body_split_strip	100	Ham	6924	30	–	Accuracy	99.15%	0:15:23
		Spam	56	2342	–	Precision	99.01%	
		Phish	13	2262	–	Recall	99.57%	
headers_short_ body_split_strip	500	Ham	6928	26	–	Accuracy	99.22%	0:25:25
		Spam	58	2340	–	Precision	99.07%	
		Phish	7	2268	–	Recall	99.63%	
headers_short_ body_split_strip	1000	Ham	6931	23	–	Accuracy	99.25%	0:28:04
		Spam	57	2341	–	Precision	99.09%	
		Phish	7	2268	–	Recall	99.67%	
headers_short_ body_split_strip	5000	Ham	6931	23	–	Accuracy	99.25%	0:28:03
		Spam	57	2341	–	Precision	99.09%	
		Phish	7	2268	–	Recall	99.67%	
full	100	Ham	6925	29	–	Accuracy	99.22%	0:42:00
		Spam	51	2347	–	Precision	99.11%	
		Phish	11	2264	–	Recall	99.58%	
full	500	Ham	6929	25	–	Accuracy	99.31%	1:16:35
		Spam	45	2353	–	Precision	99.21%	
		Phish	10	2265	–	Recall	99.64%	
full	1000	Ham	6929	25	–	Accuracy	99.34%	1:20:59
		Spam	41	2357	–	Precision	99.26%	
		Phish	11	2264	–	Recall	99.64%	
full	5000	Ham	6929	25	–	Accuracy	99.34%	1:21:04
		Spam	41	2357	–	Precision	99.26%	
		Phish	11	2264	–	Recall	99.64%	
full_short	100	Ham	6924	30	–	Accuracy	99.14%	0:15:10
		Spam	56	2342	–	Precision	99.00%	
		Phish	14	2261	–	Recall	99.57%	
full_short	500	Ham	6929	25	–	Accuracy	99.25%	0:25:06

Ham vs. Spam + Phish Results								
Feature Set	***c***	**Raw Results (Confusion Matrix)**				**Performance**		**Running Time**
			Ham	**Spam**	**Phish**			
		Spam	57	2341	–	Precision	99.11%	
		Phish	5	2270	–	Recall	99.64%	
full_short	1000	Ham	6924	30	–	Accuracy	99.18%	0:27:37
		Spam	58	2340	–	Precision	99.07%	
		Phish	7	2268	–	Recall	99.57%	
full_short	5000	Ham	6930	24	–	Accuracy	99.23%	0:27:38
		Spam	58	2340	–	Precision	99.07%	
		Phish	7	2268	–	Recall	99.65%	
full_short_ no_split	100	Ham	6927	27	–	Accuracy	99.23%	0:13:55
		Spam	50	2348	–	Precision	99.11%	
		Phish	12	2263	–	Recall	99.61%	
full_short_ no_split	500	Ham	6933	21	–	Accuracy	99.35%	0:22:52
		Spam	49	2349	–	Precision	99.21%	
		Phish	6	2269	–	Recall	99.70%	
full_short_ no_split	1000	Ham	6934	20	–	Accuracy	99.38%	0:26:03
		Spam	49	2349	–	Precision	99.26%	
		Phish	3	2272	–	Recall	99.71%	
full_short_ no_split	5000	Ham	6934	20	–	Accuracy	99.38%	0:26:03
		Spam	49	2349	–	Precision	99.26%	
		Phish	3	2272	–	Recall	99.71%	
full_short_no_ split_no_count	100	Ham	6929	25	–	Accuracy	99.29%	0:13:59
		Spam	46	2352	–	Precision	99.17%	
		Phish	12	2263	–	Recall	99.64%	
full_short_no_ split_no_count	500	Ham	6934	20	–	Accuracy	99.35%	0:23:08
		Spam	50	2348	–	Precision	99.20%	
		Phish	6	2269	–	Recall	99.71%	
full_short_no_ split_no_count	1000	Ham	6933	21	–	Accuracy	99.34%	0:27:22
		Spam	50	2348	–	Precision	99.20%	

Ham vs. Spam + Phish Results								
Feature Set	*c*	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
		Phish	6	2269	–	Recall	99.70%	
full_short_no_ split_no_count	5000	Ham	6933	20	–	Accuracy	99.35%	0:27:45
		Spam	50	2348	–	Precision	99.20%	
		Phish	6	2269	–	Recall	99.71%	

The following are the full classification results of tests in Section 4.6.

Ham vs. Spam vs. Phish Results								
Feature Set	*c*	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
trigram (body)	100	Ham	6928	23	3	Accuracy	98.90%	0:40:05
		Spam	76	2301	21	Precision	98.55%	
		Phish	26	44	2205	Recall	99.63%	
trigram (body)	500	Ham	6940	12	2	Accuracy	99.22%	1:08:05
		Spam	57	2318	23	Precision	98.90%	
		Phish	20	48	2207	Recall	99.80%	
trigram (body)	1000	Ham	6941	11	2	Accuracy	99.28%	1:30:03
		Spam	49	2326	23	Precision	98.99%	
		Phish	22	45	2208	Recall	99.81%	
trigram (body)	5000	Ham	6940	14	0	Accuracy	99.29%	2:58:51
		Spam	47	2328	23	Precision	99.02%	
		Phish	22	51	2202	Recall	99.80%	
headers_body	100	Ham	6929	21	4	Accuracy	99.31%	0:57:54
		Spam	46	2336	16	Precision	99.21%	
		Phish	9	38	2228	Recall	99.64%	
headers_body	500	Ham	6929	18	7	Accuracy	99.41%	1:45:41

Ham vs. Spam vs. Phish Results								
Feature Set	*c*	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
		Spam	38	2346	14	Precision	99.37%	
		Phish	6	35	2234	Recall	99.64%	
headers_body	1000	Ham	6928	20	6	Accuracy	99.36%	2:05:57
		Spam	42	2345	11	Precision	99.31%	
		Phish	6	32	2237	Recall	99.63%	
headers_body	5000	Ham	6928	20	6	Accuracy	99.36%	2:04:03
		Spam	42	2345	11	Precision	99.31%	
		Phish	6	32	2237	Recall	99.63%	
headers_body_ split_strip	100	Ham	6927	22	5	Accuracy	99.29%	2:18:10
		Spam	45	2337	16	Precision	99.20%	
		Phish	11	45	2219	Recall	99.61%	
headers_body_ split_strip	500	Ham	6931	18	5	Accuracy	99.32%	4:16:46
		Spam	44	2331	23	Precision	99.20%	
		Phish	12	46	2217	Recall	99.67%	
headers_body_ split_strip	1000	Ham	6927	22	5	Accuracy	99.29%	4:27:57
		Spam	45	2337	16	Precision	99.20%	
		Phish	11	45	2219	Recall	99.61%	
headers_body_ split_strip	5000	Ham	6927	22	5	Accuracy	99.29%	4:16:59
		Spam	45	2337	16	Precision	99.20%	
		Phish	11	45	2219	Recall	99.61%	
short_body	100	Ham	6929	21	4	Accuracy	98.91%	0:25:36
		Spam	79	2303	16	Precision	98.55%	
		Phish	23	43	2209	Recall	99.64%	
short_body	500	Ham	6937	12	5	Accuracy	98.92%	0:40:05
		Spam	66	2315	17	Precision	98.76%	
		Phish	21	50	204	Recall	99.76%	
short_body	1000	Ham	6938	11	5	Accuracy	99.11%	0:53:59
		Spam	64	2318	16	Precision	98.75%	
		Phish	24	47	2204	Recall	99.77%	

Ham vs. Spam vs. Phish Results								
Feature Set	*c*	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
short_body	5000	Ham	6937	13	4	Accuracy	99.19%	1:49:01
		Spam	55	2320	23	Precision	98.90%	
		Phish	22	51	2202	Recall	99.76%	
headers_short_ body	100	Ham	6927	21	6	Accuracy	99.29%	0:33:45
		Spam	50	2341	7	Precision	99.20%	
		Phish	6	25	2244	Recall	99.61%	
headers_short_ body	500	Ham	6934	14	6	Accuracy	99.38%	0:59:38
		Spam	47	2346	5	Precision	99.26%	
		Phish	5	26	2244	Recall	99.71%	
headers_short_ body	1000	Ham	6934	14	6	Accuracy	99.37%	1:13:29
		Spam	47	2348	3	Precision	99.24%	
		Phish	6	20	2249	Recall	99.71%	
headers_short_ body	5000	Ham	6934	14	6	Accuracy	99.37%	1:13:27
		Spam	47	2348	3	Precision	99.24%	
		Phish	6	20	2249	Recall	99.71%	
headers_short_ body_strip	100	Ham	6927	21	6	Accuracy	99.29%	0:35:23
		Spam	50	2341	7	Precision	99.20%	
		Phish	6	25	2244	Recall	99.61%	
headers_short_ body_strip	500	Ham	6934	14	6	Accuracy	99.38%	1:02:41
		Spam	47	2346	5	Precision	99.26%	
		Phish	5	26	2244	Recall	99.71%	
headers_short_ body_strip	1000	Ham	6934	14	6	Accuracy	99.37%	1:16:10
		Spam	47	2348	3	Precision	99.24%	
		Phish	6	20	2249	Recall	99.71%	
headers_short_ body_strip	5000	Ham	6934	14	6	Accuracy	99.37%	1:16:29
		Spam	47	2348	3	Precision	99.24%	
		Phish	6	20	2249	Recall	99.71%	
headers_short_ body_split_strip	100	Ham	6927	22	5	Accuracy	99.18%	0:40:26
		Spam	62	2328	8	Precision	99.03%	

Ham vs. Spam vs. Phish Results								
Feature Set	***c***	**Raw Results (Confusion Matrix)**				**Performance**		**Running Time**
			Ham	**Spam**	**Phish**			
		Phish	6	28	2241	Recall	99.61%	
headers_short_ body_split_strip	500	Ham	6931	16	7	Accuracy	99.28%	1:13:10
		Spam	54	2339	5	Precision	99.13%	
		Phish	7	28	2240	Recall	99.67%	
headers_short_ body_split_strip	1000	Ham	6931	17	6	Accuracy	99.27%	1:28:38
		Spam	56	2339	3	Precision	99.11%	
		Phish	6	30	2239	Recall	99.67%	
headers_short_ body_split_strip	5000	Ham	6931	17	6	Accuracy	99.27%	1:26:38
		Spam	56	2339	3	Precision	99.11%	
		Phish	6	30	2239	Recall	99.67%	
full	100	Ham	6928	23	3	Accuracy	99.22%	1:53:54
		Spam	53	2324	21	Precision	99.07%	
		Phish	12	43	2220	Recall	99.63%	
full	500	Ham	6929	20	5	Accuracy	99.28%	3:48:52
		Spam	49	2329	20	Precision	99.16%	
		Phish	10	45	2220	Recall	99.64%	
full	1000	Ham	6926	23	5	Accuracy	99.29%	4:25:44
		Spam	43	2334	21	Precision	99.23%	
		Phish	11	43	2221	Recall	99.60%	
full	5000	Ham	6926	23	5	Accuracy	99.29%	4:26:37
		Spam	43	2334	21	Precision	99.23%	
		Phish	11	43	2221	Recall	99.60%	
full_short	100	Ham	6927	23	4	Accuracy	99.17%	0:39:30
		Spam	62	2327	9	Precision	99.01%	
		Phish	7	28	2240	Recall	99.61%	
full_short	500	Ham	6931	17	6	Accuracy	99.30%	1:10:56
		Spam	51	2342	5	Precision	99.17%	
		Phish	7	30	2238	Recall	99.67%	

Ham vs. Spam vs. Phish Results								
Feature Set	***c***	**Raw Results (Confusion Matrix)**				**Performance**		**Running Time**
			Ham	**Spam**	**Phish**			
full_short	1000	Ham	6931	18	5	Accuracy	99.27%	1:26:56
		Spam	53	2341	4	Precision	99.11%	
		Phish	9	30	2236	Recall	99.67%	
full_short	5000	Ham	6931	18	5	Accuracy	99.27%	1:26:47
		Spam	53	2341	4	Precision	99.11%	
		Phish	9	30	2236	Recall	99.67%	
full_short_ no_split	100	Ham	6927	21	6	Accuracy	99.23%	0:33:07
		Spam	54	2338	6	Precision	99.10%	
		Phish	9	26	2240	Recall	99.61%	
full_short_ no_split	500	Ham	6934	14	6	Accuracy	99.36%	0:59:04
		Spam	50	2344	4	Precision	99.23%	
		Phish	4	21	2250	Recall	99.71%	
full_short_ no_split	1000	Ham	6932	16	6	Accuracy	99.31%	1:11:34
		Spam	51	2343	4	Precision	99.17%	
		Phish	7	21	2247	Recall	99.68%	
/full_short_ no_split	5000	Ham	6932	16	6	Accuracy	99.31%	1:11:30
		Spam	51	2343	4	Precision	99.17%	
		Phish	7	21	2247	Recall	99.68%	
full_short_no_ split_no_count	100	Ham	6926	22	6	Accuracy	99.29%	0:33:35
		Spam	49	2341	8	Precision	99.21%	
		Phish	6	24	2245	Recall	99.60%	
full_short_no_ split_no_count	500	Ham	6935	13	6	Accuracy	99.40%	0:59:44
		Spam	48	2346	4	Precision	99.27%	
		Phish	3	23	2249	Recall	99.73%	
full_short_no_ split_no_count	1000	Ham	6932	16	6	Accuracy	99.34%	1:13:45
		Spam	50	2343	5	Precision	99.21%	
		Phish	5	25	2245	Recall	99.68%	
full_short_no_	5000	Ham	6932	16	6	Accuracy	99.34%	1:13:36

Ham vs. Spam vs. Phish Results								
Feature Set	*c*	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
split_no_count		Spam	50	2343	5	Precision	99.21%	
		Phish	5	25	2245	Recall	99.68%	

The following are the full classification results of tests in Section 4.8 on the 2005 TREC corpus.

2005 TREC Results								
Feature Set	*c*	Raw Results (Confusion Matrix)				Performance		Running Time
			Ham	Spam	Phish			
ham vs. spam	500	Ham	2074	52	–	Accuracy	98.19%	0:10:13
		Spam	129	7745	–	Precision	94.14%	
		Phish	–	–	–	Recall	97.55%	
blind phish	500	Ham	–	–	–	Accuracy	95.13%	0:01:13
		Spam	–	–	–	Precision	–	
		Phish	111	2169	–	Recall	–	
ham vs. spam + phish	500	Ham	2063	63	–	Accuracy	98.17%	0:12:22
		Spam	96	7778	–	Precision	92.72%	
		Phish	66	2214	–	Recall	97.04%	
ham vs. spam vs. phish	500	Ham	2065	57	4	Accuracy	98.18%	0:49:00
		Spam	101	7645	128	Precision	92.68%	
		Phish	62	194	2024	Recall	97.13%	

APPENDIX C: CLASSIFIER PERFORMANCE COMPARISONS

The following is a summary of the accuracy of all three classifiers for the tests performed in Section 4.2. Entries with a long dash (–) means the feature set/classifier combination was not run. Two asterisks (**) means the Maximum Entropy classifier did not run properly and the result was thrown out.

Feature Set	Accuracy			Running Time		
	NB	MaxEnt	SVM	NB	MaxEnt	SVM
unigram	86.80%	**	93.37%	0:00:03	**	0:00:59
bigram	87.91%	**	98.45%	0:00:13	**	0:02:53
trigram	90.77%	**	98.90%	0:00:39	**	0:11:06
headers_only	95.54%	99.25%	95.61%	0:00:11	0:01:59	0:00:58
headers_body	92.98%	**	99.24%	0:01:07	**	0:16:39
headers_body_count	92.80%	**	99.24%	0:00:59	**	0:16:33
headers_body_split	88.03%	**	99.17%	0:00:48	**	0:21:30
headers_body_split_strip	87.11%	**	99.14%	0:00:57	**	0:29:46
short_body	77.54%	99.20%	98.85%	0:00:29	0:19:49	0:07:53
headers_short_body	81.02%	99.27%	99.12%	0:00:41	0:28:29	0:10:39
headers_short_body_strip	80.49%	99.52%	99.12%	0:00:40	0:28:37	0:10:40
headers_short_body_split_ strip	85.88%	99.33%	99.07%	0:00:42	0:30:39	0:11:59
full	87.63%	**	99.16%	0:01:06	**	0:30:47
full_short	86.62%	99.35%	99.04%	0:00:42	0:32:02	0:11:49
full_short_no_split	79.85%	99.39%	99.13%	0:00:41	0:27:33	0:10:33
full_short_no_split_ no_count	80.92%	99.43%	99.12%	0:00:41	0:28:18	0:10:33

The following is a summary of the accuracy of all three classifiers for the tests performed in Section 4.5. Entries with a long dash (–) means the feature set/classifier combination was not run. Two asterisks (**) means the Maximum Entropy classifier did not run properly and the result was thrown out.

Feature Set	**Accuracy**			**Running Time**		
	NB	**MaxEnt**	**SVM**	**NB**	**MaxEnt**	**SVM**
unigram	80.03%	**	–	0:00:04	**	–
bigram	85.19%	**	–	0:00:15	**	–
trigram	88.05%	**	99.17%	0:01:07	**	0:25:10
headers_only	96.08%	99.44%	–	0:00:13	0:02:34	–
headers_body	90.97%	**	99.38%	0:01:05	**	0:39:10
headers_body_count	91.01%	**	–	0:01:14	**	–
headers_body_split	87.71%	**	–	0:01:39	**	–
headers_body_split_strip	88.51%	**	99.33%	0:01:29	**	1:19:02
short_body	78.87%	99.29%	99.03%	0:00:41	0:32:22	0:15:59
headers_short_body	81.47%	99.60%	99.36%	0:01:03	0:38:28	0:23:28
headers_short_body_strip	81.47%	99.55%	99.36%	0:01:02	0:40:30	0:23:02
headers_short_body_split_ strip	87.33%	99.43%	99.22%	0:01:18	0:44:24	0:25:25
full	88.19%	**	99.31%	0:01:52	**	1:16:35
full_short	87.70%	99.37%	99.25%	0:00:56	0:43:39	0:25:06
full_short_no_split	81.02%	99.45%	99.35%	0:00:52	0:38:38	0:22:52
full_short_no_split_ no_count	80.50%	99.58%	99.35%	0:00:52	0:40:23	0:23:08

The following is a summary of the accuracy of all three classifiers for the tests performed in Section 4.6. Entries with a long dash (–) means the feature set/classifier combination was not run. Two asterisks (**) means the Maximum Entropy classifier did not run properly and the result was thrown out.

Feature Set	Accuracy			Running Time		
	NB	MaxEnt	SVM	NB	MaxEnt	SVM
unigram	75.42%	**	–	0:00:08	**	–
bigram	83.30%	**	–	0:00:16	**	–
trigram	87.64%	**	98.61%	0:01:00	**	1:08:05
headers_only	96.07%	99.35%	–	0:00:13	0:04:29	–
headers_body	91.73%	**	98.99%	0:01:15	**	1:45:41
headers_body_count	92.07%	**	–	0:01:27	**	–
headers_body_split	80.87%	**	–	0:01:34	**	–
headers_body_split_strip	83.04%	**	98.73%	0:01:51	**	4:16:46
short_body	74.63%	99.10%	98.22%	0:00:43	0:51:39	0:40:05
headers_short_body	77.81%	99.29%	99.11%	0:00:55	1:09:14	0:59:38
headers_short_body_strip	78.51%	99.50%	99.11%	0:00:55	1:07:17	1:02:41
headers_short_body_split_strip	82.03%	99.35%	98.99%	0:00:57	1:16:04	1:13:10
full	83.06%	**	98.72%	0:01:32	**	3:48:52
full_short	82.55%	99.34%	99.00%	0:01:00	1:19:51	1:10:56
full_short_no_split	77.52%	99.40%	99.15%	0:00:55	1:09:34	0:59:04
full_short_no_split_no_count	78.13%	99.36%	99.17%	0:00:55	1:10:04	0:59:44

APPENDIX D: PERFORMANCE DATA FOR P-VALUE CALCULATIONS

The following are the group classification results required for calculating the *p*-value. The accuracy is taken from the Headers + Short Body + HTML + URL feature set, with *c* = 500.

Group	Accuracy		
	Ham vs. Spam	Ham vs. Spam + Phish	Ham vs. Spam vs. Phish
01	99.359%	99.312%	99.656%
02	98.824%	98.968%	99.968%
03	99.679%	99.742%	99.742%
04	99.573%	99.398%	99.656%
05	98.825%	98.969%	98.969%
06	98.717%	99.139%	99.225%
07	99.251%	99.570%	99.226%
08	98.822%	99.139%	99.225%
09	99.144%	99.226%	98.968%
10	100.000%	100.000%	100.000%
mean	**0.99219**	**0.99346**	**0.99464**
std. dev.	**0.00434**	**0.00337**	**0.00388**
***p*-value**	**0.05695**		**0.48016**

The following are the group classification results required for calculating the *p*-value for the 2005 TREC corpus. The accuracy is taken from the Short Body + HTML + URL feature set, with *c* = 500.

Group	**Accuracy**		
	Ham vs. Spam	**Ham vs. Spam + Phish**	**Ham vs. Spam vs. Phish**
01	96.800%	96.987%	97.476%
02	98.300%	98.371%	98.371%
03	98.000%	98.697%	98.697%
04	96.900%	96.906%	96.743%
05	99.400%	99.104%	99.186%
06	98.300%	98.941%	98.453%
07	98.800%	97.801%	97.883%
08	98.800%	98.453%	98.453%
09	98.800%	98.779%	98.779%
10	97.800%	97.638%	97.720%
mean	**0.98190**	**0.98168**	**0.98176**
std. dev.	**0.00843**	**0.00792**	**0.00721**
***p*-value**	**0.95206**		**0.9805**

APPENDIX E: SOURCE CODE

This appendix contains the code I wrote to extract features and run the 10-way validation on the data. I have not chosen a license. For now, the code is available for non-commercial or research use. You may modify the code as much as you want. And, of course, there is no warranty of any kind. **If you use this code, please reference this thesis in your work. Use of this code without attributing the author is not allowed.** Commercial users, please contact the author before using this code.

feature_extractor.py is the class which extracts the features from an email message. It has several configurable properties which specify which features to extract. (Note: requires Python 3)

feature_extractor.py

```
#!/usr/bin/env python
'''
Created on Jul 5, 2011
Last modified on Sept. 11, 2011

@author: alberto
'''

import re
#import math

class EmailFeatureExtractor:
    # Class properties
    __vectors = {}
    __features = {}
    __charset = ""

    doUnigram = True
    doBigram = False
    doTrigram = False
    doHeaders = True
    doPartCounts = False
    doBody = False
    doShortBody = False
    separateHtml = False
    stripHtml = False
    doUrlFeatures = False
    #normalizeCounts = False

    # Constructor
    def __init__(self):

        # Set the pre-defined features
        self.__vectors["MESSAGE_PARSE_ERROR"] = 1
        '''
        self.__vectors["BYTE_DECODE_ERROR"] = 2
        self.__vectors["MESSAGE_IS_MULTIPART"] = 3
        self.__vectors["HAS_PLAIN_TEXT_PART"] = 4
        self.__vectors["HAS_HTML_PART"] = 5
        self.__vectors["ATTACHMENT_COUNT"] = 6
        self.__vectors["EMPTY_BODY"] = 7
        self.__vectors["EMPTY_PLAIN_TEXT_PART"] = 8
        self.__vectors["EMPTY_HTML_PART"] = 9
        self.__vectors["EMPTY_OTHER_PART"] = 10

        # Set the phish features
        self.__vectors["HAS_RAW_URL"] = 11
        self.__vectors["NON_MATCHING_RAW_URL"] = 12
        self.__vectors["USES_CLICK_HERE"] = 13
        self.__vectors["URL_USES_IP"] = 14
        self.__vectors["LINK_COUNT"] = 15
        self.__vectors["LINK_DOMAIN_COUNT"] = 16
        '''


    def getVectors(self):
        return self.__vectors


    def extract(self, message):
        # See what character set we have; if not, set a default
        if message.get_charset() != None:
            self.__charset = message.get_charset()
        else:
            self.__charset = "iso-8859-1"

        # Initialize the feature list for this message
        self.__features = {}

        # See if we are going to do headers
        if self.doHeaders:
            if "Subject" in message:
                self.__extractNgrams("SUBJECT_", message["Subject"])

            if "Return-Path" in message:
                self.__extractNgrams("RETURN_PATH_", message["Return-Path"])

            if "From" in message:
                self.__extractNgrams("FROM_", message["From"])

            if "To" in message:
                self.__extractNgrams("TO_", message["To"])

            if "Mailing-List" in message:
                self.__extractNgrams("MAILING_LIST_", message["List-Id"])

            if "List-Id" in message:
                self.__extractNgrams("LIST_ID_", message["List-Id"])

        # See if we are doing anything with the body
        if self.doPartCounts or self.doBody or self.doShortBody \
        or self.doUrlFeatures:
            # Extract features from the text parts of the message
            for part in message.walk():
                # See if we have a multipart container
                if part.get_content_maintype() == 'multipart':
                    if self.doPartCounts:
                        self.__increase_feature_count("MESSAGE_IS_MULTIPART")
                    continue

                # Skip if this part is not text
                if part.get_content_maintype() != "text":
                    if self.doPartCounts:
                        self.__increase_feature_count("ATTACHMENT_COUNT")
                    continue

                # See what kind of text part we have
                content_type = ""
                if part.get_content_subtype() == "plain":
                    if self.doPartCounts:
```

```
                    self.__increase_feature_count("HAS_PLAIN_TEXT_PART")
                if self.separateHtml:
                    content_type = "PLAIN_TEXT"
            elif part.get_content_subtype() == "html":
                if self.doPartCounts:
                    self.__increase_feature_count("HAS_HTML_PART")
                if self.separateHtml:
                    content_type = "HTML"
            else:
                # Consider this an attachment
                if self.doPartCounts:
                    self.__increase_feature_count("ATTACHMENT_COUNT")
                continue

            # Get the text and decode it
            text = part.get_payload()
            if part.get_content_charset() != None:
                self.__charset = part.get_content_charset()
            if isinstance(text, bytes):
                try:
                    text = text.decode(self.__charset)
                except:
                    self._increase_feature_count("BYTE_DECODE_ERROR")
                    text = text.decode(self.__charset, "ignore")

            # See if the text part is empty
            if len(text.strip()) == 0:
                self.__increase_feature_count("EMPTY_" + content_type +
                                              "_PART")
                continue

            # See if we are extracting features from the body
            if self.doBody:
                # Extract the ngrams
                self.__extractNgrams("BODY_" + content_type + "_", text)

                # See if we need to do stripped HTML
                if content_type == "HTML" and self.stripHtml:
                    p = re.compile(r'<.*?>')
                    s_text = p.sub('', text)
                    self.__extractNgrams("BODY_S" + content_type + "_",
                                         s_text)

            # See if we are extracting features from the shortened body
            if self.doShortBody:
                # See if the length is 2048 or larger
                if len(text) >= 2048:
                    # See which features we need to extract
                    self.__extractNgrams("BODY_START_" + content_type + "_",
                                         text[:1024])
                    self.__extractNgrams("BODY_END_" + content_type + "_",
                                         text[-1024:])
                else:
                    # Extract features from the whole text
                    self.__extractNgrams("BODY_" + content_type + "_", text)

                # See if we need to do stripped HTML
                if type == "HTML" and self.stripHtml:
                    p = re.compile(r'<.*?>')
                    s_text = p.sub('', text)

                    if len(s_text) >= 2048:
                        self.__extractNgrams("BODY_START_S" +
                                             content_type + "_",
                                             s_text[:1024])
                        self.__extractNgrams("BODY_END_S" + content_type +
                                             "_", s_text[-1024:])
                    else:
                        self.__extractNgrams("BODY_S" + content_type + "_",
                                             s_text)

            if self.doUrlFeatures:
                if part.get_content_subtype() == "html":
                    self.__extractUrlFeatures(text)

    return self.__features


def __extractNgrams(self, vector_prefix, text):
    if self.doUnigram:
        self.__extractNgramsBySize(1, vector_prefix + "1_", text)

    if self.doBigram:
        self.__extractNgramsBySize(2, vector_prefix + "2_", text)

    if self.doTrigram:
        self.__extractNgramsBySize(3, vector_prefix + "3_", text)


def __extractNgramsBySize(self, size, vector_prefix, text):
    # Make sure we have either text or bytes for the text
    if not isinstance(text, str) and not isinstance(text, bytes):
        return

    # See if we need to decode bytes
    if isinstance(text, bytes):
        try:
            text = text.decode(self.__charset)
        except:
            self.__increase_feature_count("BYTE_DECODE_ERROR")
            text = text.decode(self.__charset, "ignore")

    # Make sure we have a size greater than 0
    if (size <= 0):
        return

    # Make sure the length of the text is greater than our ngram size
    if len(text) < size:
        # TODO: allow it to work with small amounts of text
        return

    # If size is greater than 1, begin extraction with start symbols
    for i in range(1, size):
        # Initialize the vector name
        vector = vector_prefix

        # Add the start symbols
        for j in range(0, size - i):
            vector += "[start]"

        # Add the start of the text
        vector += text[0:i]

        # Add the feature
        self.__increase_feature_count(vector)

    for i in range(0, len(text) - size + 1):
        # Create the vector
        vector = vector_prefix + text[i:i+size]

        # Add this feature
        self.__increase_feature_count(vector)

    # Add the stop symbol
    if size > 1:
        vector = vector_prefix + text[1 - size:] + "[stop]"

        # Add the feature
        self.__increase_feature_count(vector)

    # See if we are normalizing the counts
    #if self.normalizeCounts:
        # Get the total number of features
        #total = 0;
        #for count in features:
        #    total += count
        #norm_factor = total

    #    sum_sq = 0
    #    for count in features:
    #        sum_sq += count * count
    #    norm_factor = math.sqrt(sum_sq)

        # Adjust the entries
    #    for feature in features.keys():
    #        self.__features[feature] = \
    #            round(features[feature] / norm_factor, 6)
    #else:
    #    for feature in features.keys():
    #        self.__features[feature] = features[feature]

def __extractUrlFeatures(self, text):
    # Get the link tags
    links = re.findall('''<a href=["'](http.[^"']+)["']>(.[^<>]*)</a>''', \
                       str(text), re.I)
    if len(links) > 0:
        # Store the number of links
        self.__increase_feature_count("LINK_COUNT", len(links))

        # Create the list of unique domains
        domain_list = {}

        # Go through each link
        for link in links:
            # See if the text of the link is a URL
            url = re.search('''(http|https)://.+''', link[1], re.I)
            if url:
                # We have raw URLs
                self.__increase_feature_count("HAS_RAW_URL")

                # See if the raw URL matches the href link
                if url.group().find(link[0]) >= 0:
                    self.__increase_feature_count("NON_MATCHING_RAW_URLS")
            else:
                # See if the link text has HERE
                if re.search('''h.*e.*r.*e''', link[1], re.I):
                    self.__increase_feature_count("USES_CLICK_HERE")

            # See if the href link uses IP address
            if re.search('''[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.''' +
                         '''[0-9]{1,3}''', link[0]):
                self.__increase_feature_count("URL_USES_IP")

            # Add the domain to the list
            url_split = link[0].split("/")
            domain_list[url_split[2]] = True

        # Add the count of distinct domains
        self.__increase_feature_count("LINK_DOMAIN_COUNT", len(domain_list))


def __get_vector_number(self, vector):
    if vector not in self.__vectors:
        self.__vectors[vector] = len(self.__vectors) + 1
    return self.__vectors[vector]
def __increase_feature_count(self, vector, count = 1):
    vector_number = self.__get_vector_number(vector)

    if vector_number in self.__features:
        self.__features[vector_number] += count
    else:
        self.__features[vector_number] = count
```

extract_features.py is the main executable script that loads the email messages from the corpora and feeds them into the feature extractor class. It requires three parameters: the directory where the corpora is located, where the features will be written to, and the model (or feature set) that will be used (run the program with no parameters for usage). The corpora directory needs to have subdirectories with the different categories. Inside each category the data needs to be split into directories which divide the data. In this case, the data was divided into 10 groups. (You can change the categories and groups by changing the definition of the *corpus_dir* and *dir_list* arrays.) This program will recursively (Note: requires Python 3)

extract_features.py

```
#!/usr/bin/env python
'''
Created on Jul 5, 2011

@author: alberto

This script will extract features from email corpora in 10-way split
directories. It will save a file for each of the 10 directories based on the
model name given.
'''

# Imports
import sys
import os
import email
from feature_extractor import EmailFeatureExtractor

# Get the paths to the data from the command line arguments
corpora = None
feature_dir = None
model = "default"

# Initialize the feature extractor
extractor = EmailFeatureExtractor()

# Go through the arguments
for raw_arg in sys.argv[1:]:
    # Split the argument on =
    arg = raw_arg.split("=", 2)
    if (len(arg) == 2):
        # See which argument we have
        if (arg[0] == "--corpora"):
            if (os.path.isdir(arg[1])):
                corpora = arg[1]
        elif (arg[0] == "--features"):
            if (os.path.isdir(arg[1])):
                feature_dir = arg[1]
        elif (arg[0] == "--model"):
            # See which kind of model we are doing:
            if arg[1] == "unigram":
                model = "unigram"
                extractor.doUnigram = True
                extractor.doBigram = False
                extractor.doTrigram = False
                extractor.doHeaders = False
                extractor.doPartCounts = False
                extractor.doBody = True
                extractor.doShortBody = False
                extractor.separateHtml = False
                extractor.stripHtml = False
                extractor.doUrlFeatures = False
            elif arg[1] == "bigram":
                model = "bigram"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = False
                extractor.doHeaders = False
                extractor.doBody = True
                extractor.doPartCounts = False
                extractor.doShortBody = False
                extractor.separateHtml = False
                extractor.stripHtml = False
                extractor.doUrlFeatures = False
            elif arg[1] == "trigram":
                model = "trigram"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = False
                extractor.doPartCounts = False
                extractor.doBody = True
                extractor.doShortBody = False
                extractor.separateHtml = False
                extractor.stripHtml = False
                extractor.doUrlFeatures = False
            elif arg[1] == "headers_only":
                model = "headers_only"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = True
                extractor.doPartCounts = False
                extractor.doBody = False
                extractor.doShortBody = False
                extractor.separateHtml = False
                extractor.stripHtml = False
                extractor.doUrlFeatures = False
            elif arg[1] == "headers_body":
                model = "headers_body"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = True
                extractor.doPartCounts = False
                extractor.doBody = True
                extractor.doShortBody = False
                extractor.separateHtml = False
                extractor.stripHtml = False
                extractor.doUrlFeatures = False
            elif arg[1] == "headers_body_count":
                model = "headers_body_count"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = True
                extractor.doPartCounts = True
                extractor.doBody = True
                extractor.doShortBody = False
                extractor.separateHtml = False
                extractor.stripHtml = False
                extractor.doUrlFeatures = False
            elif arg[1] == "headers_body_split":
                model = "headers_body_split"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = True
                extractor.doPartCounts = True
                extractor.doBody = True
                extractor.doShortBody = False
                extractor.separateHtml = True
                extractor.stripHtml = False
                extractor.doUrlFeatures = False
            elif arg[1] == "headers_body_split_strip":
                model = "headers_body_split_strip"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = True
                extractor.doPartCounts = True
                extractor.doBody = True
                extractor.doShortBody = False
                extractor.separateHtml = True
                extractor.stripHtml = True
                extractor.doUrlFeatures = False
            elif arg[1] == "short_body":
                model = "short_body"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = False
                extractor.doPartCounts = False
                extractor.doBody = False
                extractor.doShortBody = True
                extractor.separateHtml = False
                extractor.stripHtml = False
                extractor.doUrlFeatures = False
            elif arg[1] == "headers_short_body":
                model = "headers_short_body"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = True
                extractor.doPartCounts = True
                extractor.doBody = False
                extractor.doShortBody = True
                extractor.separateHtml = False
                extractor.stripHtml = False
                extractor.doUrlFeatures = False
            elif arg[1] == "headers_short_body_strip":
                model = "headers_short_body_strip"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = True
                extractor.doPartCounts = True
                extractor.doBody = False
                extractor.doShortBody = True
                extractor.separateHtml = False
                extractor.stripHtml = True
                extractor.doUrlFeatures = False
            elif arg[1] == "headers_short_body_split_strip":
                model = "headers_short_body_split_strip"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = True
                extractor.doPartCounts = False
                extractor.doBody = False
                extractor.doShortBody = True
                extractor.separateHtml = True
                extractor.stripHtml = True
                extractor.doUrlFeatures = False
            elif arg[1] == "full":
                model = "full"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = True
                extractor.doPartCounts = True
                extractor.doBody = True
                extractor.doShortBody = False
                extractor.separateHtml = True
                extractor.stripHtml = True
                extractor.doUrlFeatures = True
            elif arg[1] == "full_short":
                model = "full_short"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = True
                extractor.doPartCounts = True
                extractor.doBody = False
                extractor.doShortBody = True
                extractor.separateHtml = True
                extractor.stripHtml = True
                extractor.doUrlFeatures = True
            elif arg[1] == "full_short_no_split":
                model = "full_short_no_split"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = True
                extractor.doPartCounts = True
                extractor.doBody = False
                extractor.doShortBody = True
                extractor.separateHtml = False
                extractor.stripHtml = True
                extractor.doUrlFeatures = True
            elif arg[1] == "full_short_no_split_no_count":
                model = "full_short_no_split_no_count"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = True
                extractor.doPartCounts = False
                extractor.doBody = False
                extractor.doShortBody = True
                extractor.separateHtml = False
                extractor.stripHtml = True
                extractor.doUrlFeatures = True
            elif arg[1] == "full_short_no_split_no_headers":
                model = "full_short_no_split_no_headers"
                extractor.doUnigram = True
                extractor.doBigram = True
                extractor.doTrigram = True
                extractor.doHeaders = False
                extractor.doPartCounts = True
                extractor.doBody = False
                extractor.doShortBody = True
                extractor.separateHtml = False
                extractor.stripHtml = True
                extractor.doUrlFeatures = True
            else:
                print("Unknown model: " + arg[1])
                exit(1)

if (corpora == None) or (feature_dir == None):
    print("Error Invalid arguments.")
    print("")
    print ("Usage: " + sys.argv[0] + \
        " --corpora=/path/to/split/corpora" + \
        " --features=/path/to/split/features --model=model_name" + \
        " --model=model")
    exit(1)

print("Using " + model + " model")

# Define the corpus directories
corpus_list = ["ham", "spam", "phish"]
dir_list = ["01", "02", "03", "04", "05", "06", "07", "08", "09", "10"]

# Go through the corpus directories
class_id = 0

for corpus in corpus_list:
    # Increase the class id value
    class_id += 1

    # Go through the directories
    for dir in dir_list:
        # Make sure this directory exists
        path = corpora + "/" + corpus + "/" + dir
        save_path = feature_dir + "/" + model + "/"
        if not (os.path.isdir(path)):
            print("Error: " + path + " directory does not exist")
            exit(1)

        if not (os.path.isdir(save_path)):
            os.mkdir(save_path)

        print("Extracting features in " + corpus + "/" + dir + " ...")

        # Open the file where the features will be stored
        features_fp = open(save_path + "features." + corpus + "." + str(dir),
                           "w")

        # Go through each file in the directory
        for file in os.listdir(path):
            # Load the email
            fp = open(path + "/" + file, "rb")
            parse_ok = False
            #print("  "+ file)
            try:
                message = email.message_from_binary_file(fp)
                parse_ok = True

            except:
                print("Could not parse " + corpus + "/" + file)
                features = {}
                features[1] = 1;

            # Get the features
            if parse_ok:
                features = extractor.extract(message)

            # Write out the features
            features_fp.write(str(class_id))
            for feature_number in sorted(features.keys()):
                features_fp.write(" " + str(feature_number) + ":" + \
                              str(features[feature_number]))
            features_fp.write(" # " + corpus + "/" + dir + "/" + file + "\n")
            #features_fp.write("\n")

            fp.close

        # Close the file
        features_fp.close()

#import pprint
#pprint.pprint(extractor.getVectors())
```

The following script is an example of the shell scripts I used to run the 10-way cross-validation. This particular script ran the three-way cross-validation. It shows how the data was combined from the feature extractor, combined and fed to svm_multiclass_learn to create the model. From there, the data was processed with svm_multiclass_classify and the script tallied the results.

There were four of these scripts, one for each phase of testing. They are mostly the same, except on how they consolidated the features. For example, the two-way filter would use `sed` to change the category id of 3 to 2 for all phishing samples.

Sample 10-way validation script

```
#!/bin/bash

# This script will run a 10-way validation on ham and spam using the unigram +
#  bigram + trigram model. It will also consider phish but change its class
#  identifier to spam to test wanted vs. unwanted.

corpora_dir="corpora"
tmp_dir="tmp"
svm_dir="svm"
model="test"
classes="ham phish spam"
groups="01 02 03 04 05 06 07 08 09 10"
results_ham[1]=0
results_ham[2]=0
results_ham[3]=0
results_phish[1]=0
results_phish[2]=0
results_phish[3]=0
results_spam[1]=0
results_spam[2]=0
results_spam[3]=0

# Make sure things exist
if [ ! -e "$corpora_dir/ham/$model.tokens.01" ]; then
       echo "Could not find tokens"
       exit
fi

# Go through each group
for run in $groups; do
       echo "Performing run #$run"

       # Delete the files in the temp directory
       for file in $tmp_dir/*; do
              rm $file
       done

       # Go through each class and combine the tokens
       for class in $classes; do
              echo " Combining tokens in class $class"

              # Go through each group
              for group in $groups; do
                     # See if this group is the same as the run
```

```
                        if [ "$group" == "$run" ]; then
                                cat "$corpora_dir/$class/$model.tokens.$group" > \
                                        "$tmp_dir/$class.$model.validation"
                        else
                                cat "$corpora_dir/$class/$model.tokens.$group" >> \
                                        "$tmp_dir/$model.train"
                        fi
                done
        done

        # Training the SVM model
        echo " Training the SVM model"
        "$svm_dir/svm_multiclass_learn" -c 5000 -v 0 "$tmp_dir/$model.train" \
                "$tmp_dir/$model.svm_model"

        # Go through each class
        for class in $classes; do
                echo "  Testing $class"
                "$svm_dir/svm_multiclass_classify" -v 1 \
                        "$tmp_dir/$class.$model.validation" "$tmp_dir/
$model.svm_model" \
                        "$tmp_dir/classification_results"

                # Go through the classes
                for i in 1 2 3; do
                        count=`cut -d " " -f 1 "$tmp_dir/classification_results"
| \
                                grep $i | wc -l`
                        case $class in
                                "ham" )
                                        results_ham[$i]=$(( ${results_ham[$i]} +
$count ))
                                        ;;
                                "phish" )
                                        results_phish[$i]=$(( ${results_phish[$i]} +
$count ))
                                        ;;
                                "spam" )
                                        results_spam[$i]=$(( ${results_spam[$i]} +
$count ))
                                        ;;
                        esac
                done
        done
done

# Print the results
echo ""
echo "-----===== RESULTS =====-----"
echo "       ham     phish   spam"
echo "ham    ${results_ham[1]}    ${results_ham[2]}      ${results_ham[3]}"
echo "phish  ${results_phish[1]}      ${results_phish[2]}  ${results_phish[3]}"
echo "spam   ${results_spam[1]}      ${results_spam[2]}      ${results_spam[3]}"
```