An Evaluation of Machine Learning Techniques for Enterprise Spam Filters

Andrew Tuttle, Evangelos Milios, Nauzer Kalyaniwalla

January 6, 2004

Contents

List of Tables

List of Figures

Abstract

Like a distributed denial-of-service attack, the barrage of spam email is overwhelming enterprise network resources. We propose and evaluate an architecture for a practical enterprise spam filter that provides personalized filtering on the server side using machine learning algorithms. We also introduce a novel experimental methodology that overcomes the "privacy barrier", making it possible to evaluate spam classifiers on a variety of individual, complete streams of real email. Our tests yield convincing evidence that these algorithms can be used to build practical enterprise spam filters. We show that the proposed architecture will likely be efficient and scalable. We show that the filters can be, on average, highly effective even at very low false positive rates. And we show that the algorithms offer a well-behaved tuning mechanism that can be used to manage the overall enterprise risk of legitimate mail loss.

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

We would also like to thank Dr. Sedgwick, for his critical analyses and recommendations.

Special thanks to Chris Jordan, Carole Poirier, Menen Teferra and Dr. Carolyn Watters help with the experiments. Also thanks to Jeff Allen for helping with the use of the mail server.

Chapter 1

Introduction

This thesis studies the viability of enterprise spam filters that use machine learning algorithms to recognize spam. It introduces a scalable architecture for such filters, and a novel experimental methodology for evaluating their effectiveness. Using this methodology, it overcomes the privacy barrier to evaluate three popular spam classification algorithms on complete sets of real email. The goal is to determine if these kinds of filters can be both effective and efficient enough for practical deployment in large networks.

Spam, sometimes more formally known as “Unsolicited Commercial Email” (UCE) or “Unsolicited Bulk Email” (UBE), has grown from being a minor nuisance into a global menace. Millions of email users world-wide waste untold thousands of hours every day sorting through incoming mailboxes cluttered with unwanted or even offensive messages. Meanwhile, the growth in network resources consumed by spam continues at an unsustainable rate. The spiraling direct and indirect costs of spam threaten the reliability and utility of the email infrastructure that has become so vital to the global economy.

At the core of the spam problem is the fact that spam allows vendors to transfer the vast majority of their marketing costs to the people they are marketing to. Even though the response rate to spam is believed to be vanishingly small, it is still large enough to overcome the even smaller costs of sending spam. Therefore, it becomes a volume-driven business - the more spam you send, the more money you make. But the real cost of the ensuing flood of junk email is not small, and it is borne by the owners of the networks and servers that must deliver the spam, and by the individual recipients who must waste valuable time sorting through it.

A variety of legal, structural, and technical solutions have been proposed; most are controversial and many suffer from glaring weaknesses. Legal solutions face daunting jurisdiction problems. Structural solutions, such as replacing SMTP, face staggering infrastructure costs. Many technical solutions are extremely high-maintenance, resulting in a never-ending “arms race” between the elusive spammers and the network administrators attempting to find new ways to block them.

While the world searches for a long-term solution, there is an urgent need for something businesses can use to protect their networks from the escalating costs of spam. This research is motivated by that need.

We define an “enterprise spam filter” as a system that blocks spam at the outside edge of a network shared by up to thousands of email users. Like a firewall, this system must protect the network from harmful external attack, while allowing legitimate activities to proceed unimpeded. We further define the five most important requirements of an enterprise spam filter:

- Automatic: the filter must be easy (ideally transparent) to use;
- Adaptive: the filter must require little maintenance (ideally none) to keep up with the changing nature of spam and the tricks used by spammers to evade filters;
- Effective: the filter must block the vast majority of incoming spam, close enough to the edge of the network to minimize (ideally eliminate) resource consumption;
- Safe: the filter must have a very low rate of false positives (ideally zero);
- Efficient: the filter must be both fast and scalable enough to run on shared servers.

There are a number of compelling reasons to think that machine-learning-based spam classification algorithms can be used to build such a filter. First, they automatically learn how to distinguish spam from legitimate messages. Second, they easily adapt, through periodic retraining, to changes in mail characteristics. Third, they appear to be very effective, based on a number of previous studies that have shown very promising results. Fourth, they enable each user to have their own personalized filter, precisely tuned to the kind of mail they receive. It is quite reasonable to expect a custom filter to perform better than a generic one, and be more difficult to systematically circumvent.

There are however a number of unanswered questions about the suitability of these techniques. First, they are very processor-intensive, so are typically confined to client-side solutions. Can they scale up to run on the servers of large networks? Second, they have really only been tested on small or unrealistic datasets, because of the privacy barrier. Will they perform well on large numbers of real sets of email?

Third, they offer no guarantees of avoiding false positives. Can they be made safe enough to deploy?

This thesis attempts to answer those questions, and make a convincing argument that a practical enterprise spam filter can be built using machine learning algorithms.

The specific contributions of this report are:

- A proposed architecture for efficient and scalable enterprise spam filters that use machine-learning-based spam classification algorithms;
- A novel experimental methodology that overcomes the privacy barrier - making it possible to systematically evaluate spam classification algorithms on the email received by large numbers of users;
- The first evaluation, under that methodology, of three popular spam classification algorithms - yielding arguably the most useful performance results to date for those algorithms.
- Statistical evidence of the importance of personalization in the evaluation of spam classification algorithms;
- An empirical quantification of the inherent trade-off between the effectiveness and safety of those algorithms - showing how they enable enterprises to manage their risk in a disciplined way;
- A comparison of average algorithm performance versus corpus size - showing how many messages these algorithms typically need to see before they become useful;
- A comparison of average algorithm execution times - providing evidence that the proposed architecture is indeed efficient and scalable;
- An extensible C++ API for spam classification - supporting future studies based on the new methodology.

Chapter 2 will provide the necessary background and a critical analysis of related work. Chapter 3 will define the problem and introduce the proposed solution. Chapter 4 will explain the experimental design and the software created to implement it. Chapter 5 will present and analyze the results of the experiments. Chapter 6 will highlight the conclusions drawn from the study, and propose future work.

Chapter 2

Background and Related Work

This chapter introduces the spam classification problem, highlights some of the key technical issues and discusses related work. It concludes by identifying critical questions that have not yet been adequately addressed in the literature.

2.1 Spam Classification

Text classification is a well-established area of research within the field of machine learning. At a high level, the inputs to a text classification problem are:

- a set of labeled training documents (the "training set");
- a set of labeled test documents (the "test set"); and
- a classification algorithm (the "classifier").

The classifier uses the training set to learn how to associate labels with documents. The learning mechanism may be statistical, geometrical, rule-based or something else. The trained classifier is used to predict the label of each document in the test set. Since the correct labels are already known, the classifier can be scored based on its accuracy. This is known as a *supervised learning* problem.

When evaluating the success of an algorithm in a supervised learning problem, it is critical that the training set and test set do not overlap. Many machine learning techniques are prone to *overfitting* of the training set, meaning that they can become much too reliant on the details of the specific documents in the training set, and cannot perform as well on new documents that have not been seen before. Therefore, the performance of a classifier on the training set is not considered to be a reliable indicator of the classifier's *generalization* performance.

In recent years, a number of text classification studies have focused on email documents [4, 15], typically exploring the problem of categorizing messages into a set of folders. Even more recently, the spam problem has motivated researchers to focus on the more specific problem of recognizing spam.

There are three important reasons why the body of spam classification research is distinct from the more general email classification literature. First, spam classification is strictly a binary problem; we are interested in determining if an incoming message is "spam", or "not-spam". Second, spam classification has extremely unbalanced error costs; the cost of losing a legitimate message is far greater than that of allowing a spam message to sneak into the inbox. Third, spam classification has the added challenge of contending with an adversary; the people creating the spam messages are deliberately trying to confuse your classifier!

2.2 Document Representation

Email documents are considered semi-structured text documents, consisting of structured fields with relatively well-defined semantics (the message headers), and unstructured, variable-length fields of free text (the subject line and body) [6]. The relative value of the information found in the two sections is often of interest to researchers.

Each document is typically represented by an *instance* vector - an array of *attribute* or *feature* values. This is generally known as the "Vector Space Model", or more informally, the "bag of words" approach.

Each attribute or feature represents a particular measurement made on the document, such as how many times a particular *word*, *term*, or *token* appeared. The most popular types of attributes in the spam literature are:

- Binary - the presence or absence of a token;
- Term Frequency (TF) - the number of occurrences of a token [7, 20];
- TF/IDF - TF scaled by the Inverse Document Frequency (IDF), having the effect of increasing the weight of tokens that are not common in many documents [7].

Binary attributes are by far the most common type found in the spam literature, partly because they make many calculations far more efficient.

Before training, the document representation must be decided on. This process involves the steps of "feature extraction" and "feature selection". Then, during training and testing, each document must be mapped to an instance based on the chosen document representation. This second process involves feature extraction and "instance mapping". These steps are described below.

2.3 Feature Extraction

Feature extraction is the task of parsing a document into a list of tokens. These tokens are typically individual words, tokenized by treating a certain set of ASCII characters as word delimiters. The choice of that set of delimiters is important! For example, the way you tokenize email addresses, URLs and IP addresses could have a significant impact on your results.

Some studies have looked at the effects of including "heuristic" features such as whether or not the email has an attachment, or how many consecutive exclamation marks are found in the subject line [16, 6, 9].

2.4 Feature Selection

Feature selection takes the set of all tokens found in all training documents, and decides which ones to include in the *dictionary*. A trivial feature selection process might simply decide to use all tokens found in all documents. There are, however, a number of good reasons to be more selective:

- Dimensionality reduction - extremely long instance vectors often cause extremely long execution times;
- Resistance to overfitting - some algorithms are more susceptible to overfitting for large, sparse feature spaces that have a lot of redundant features [20];
- Improved effectiveness - some algorithms perform better when only the most "useful" features are considered, with useless "noise" features rejected [7].

Typically, a potential feature set in the tens of thousands will be pared down to a few hundred or a few thousand of the most useful ones. A variety of techniques have been employed to accomplish this:

- Stop lists - reject useless words such as "the" and "a" [7, 2, 1];
- Stemming - reduce word forms to their base, for example change "running" to "run" [13, 7, 2, 1];
- Minimum term frequency - reject as uninformative words that appear in less than a minimum number of documents [16];

- Mutual information - a calculation based on information gain or entropy that identifies the tokens that most closely associated with a specific class [16, 2, 1, 8, 20].

2.5 Instance Mapping

We define “instance mapping” as the process of mapping a document to an instance vector, using the dictionary and the list of tokens found in the document. For each token in the dictionary, an attribute value is calculated. In the simplest case, binary, the value of the first attribute will be “true” if the first token in the dictionary is found anywhere in the document.

2.6 Classification

Classification is the process of predicting the label of the document represented by a given instance vector. Some classifiers output “hard” decisions - simply the predicted label. Others output “soft” decisions - typically a score representing how confident the classifier is that the given instance represents a spam message. This second group of classifiers is attractive, because they allow researchers to use thresholds to decide on the final classification. By varying the threshold, tradeoffs can be made between the two different types of errors, false positive and false negative.

2.7 Algorithms

A rich variety of text classification algorithms have been applied to the spam classification problem. These include:

- Naive Bayes [16, 13, 1, 2, 6, 9, 8, 20];
- Support Vector Machines (SVM) [7, 10, 8, 11];
- Decision trees [6, 9, 8];
- Boosted decision trees [7, 3];
- k-Nearest-Neighbor (kNN) [2, 9, 8];
- Stacked classifiers [17];
- Rule learners [7, 9, 8];

- Rocchio [7]; and,
- Centroids [21].

2.8 Experimental Methodologies

A common challenge in classification research is a lack of large quantities of accurately labelled data. This is especially true in spam classification research, because most people consider their email private, and aren't particularly enthused by the idea of giving it to researchers to study. In general though, it is expensive to have human "experts" go through thousands of documents and give each one the correct label. This challenge is magnified by the need to have sufficient data to be able to maintain separate training and test sets. As a result, it could be difficult to obtain statistically significant results.

Fortunately, a process called "cross validation" makes it possible for researchers to take maximum advantage of limited training data. A common version of the method is known as 10-fold cross validation. Using this methodology, the complete set of labelled documents is divided into 10 random folds (often *stratified*, meaning that each fold has approximately the same class distribution as the overall set). Then, for each fold, all the other 9 folds are used for training, and the current fold is used for testing. This process ensures that every document can be used for testing, but is never tested by a classifier that also saw it during training. This process is repeated ten times, to generate even more results.

In the spam classification literature, the privacy barrier has forced researchers to find alternative sources of data. One popular example is the "LingSpam" corpus [2, 1, 9, 8] - a collection of postings from a mailing list for linguists, combined with a number of spam email messages. The presumption is that this corpus resembles real email strongly enough to produce relevant results. Other researchers have tested on their own email [13], or a pooled collection of messages from themselves and a few colleagues [16, 7]. Few studies kept messages from different people separate [6], or made sure to include legitimate messages that people ordinarily delete [16].

Furthermore a surprisingly large proportion of the studies did not use cross validation; many of them instead chose to divide, based on chronological order, the messages into a single training set and a single test set. While this methodology sounds intuitive, the results obtained from it are not the most statistically sound that can be obtained from the available data.

2.9 Metrics

Virtually all classifier evaluation metrics used in the spam literature are based on the "confusion matrix". This divides classification decisions into one of four quadrants:

- True positives - correctly identified spam messages;
- False positives - legitimate messages incorrectly identified as spam;
- True negatives - correctly identified legitimate messages;
- False negatives - spam messages incorrectly identified as legitimate.

Recall measures the proportion of the members of a particular class that were identified correctly. For example, spam recall is the proportion of spam that the classifier successfully recognized.

Precision measures the proportion of instances predicted to belong to a particular class that really belong there. For example, for all those messages classified as spam, what proportion of them really are? That is spam precision.

False positive rate measures the proportion of legitimate messages that get misclassified as spam.

Accuracy measures the proportion of all messages that were correctly classified.

Error rate measures the proportion of all messages that were incorrectly classified.

The F measure is a handy way to combine recall and precision into a single measure of "goodness".

The Total Cost Ratio (TCR) [1] is another single measure, one that takes into account the asymmetric costs of errors.

2.10 Cost Sensitivity

Much of the focus in the spam classification literature is on cost-sensitivity. Given the highly unbalanced costs of classification errors, it is intuitive to think that any evaluation should favor algorithms that tend to make less costly errors. A variety of approaches to this have been attempted, with the TCR mentioned above being one of the most popular. One problem is that under certain assumptions, a single false positive makes a classifier score worse than a trivial one that simply calls every message legitimate [1]. Some studies used different thresholds to model specific cost scenarios [16, 6, 3].

Some researchers have attempted to introduce cost sensitivity into the training process by biasing their classifiers against false positives [9, 10, 8].

2.11 Summary

The existing body of spam classification research collectively shows very promising results. It appears that several machine learning algorithms - especially Naive Bayes, support vector machines and boosted decision trees - are capable of achieving a high spam recall at a low false positive rate. The key issues of algorithm identification, feature selection, and cost sensitivity have all been thoroughly examined.

That body of research, however, has little to say about whether machine-learning-based spam classifiers can be used to build effective and efficient enterprise spam filters. Enterprise spam filters must serve hundreds or thousands of users, all of whom get different kinds of email, and in different proportions. All of that mail must be filtered by shared servers at the edge of the network, to prevent spam from flooding the local network. To convince someone that spam classifiers can function in this kind of environment, you need to prove to them that the filters will work for everyone, not just computer scientists and linguists! And you need to prove to them that spam classifiers can handle the load. These questions have not yet been adequately addressed.

The studies to date have not been able to run experiments on realistic data - a variety of complete individual email streams - because it simply hasn't been available. The privacy barrier has forced people to use just their own mail, or a collection of mail contributed by a few close colleagues, or one of a few publicly-available corpora of questionable value. Few of them have even tested on complete email streams; most seem to make do with just the messages that were not already deleted. Results based on this kind of data are not convincing enough.

The studies to date have also not produced much evidence that machine-learning-based spam classifiers can be efficient enough to run on the server. This is critical, because client-side filters will not solve the problem. Many classification algorithms are notoriously slow. Training and classification times must be studied, as well as disk space and bandwidth requirements.

These are the issues that this report will address. The next chapter will explain how.

Chapter 3

Problem Definition and Proposed Solution

This chapter defines the questions that this report will ask, and proposes how to answer them.

3.1 Unanswered Questions

The goal of this report is to provide a convincing argument that machine-learning-based spam classification algorithms can be used to build practical enterprise spam filters. In Chapter 1, we listed the five most important requirements of such filters: they must be automatic, adaptive, effective, safe and efficient.

We know that these algorithms are inherently automatic and adaptive. What we don't know is whether they can be effective and safe enough to use on thousands of different real email streams, and whether they can handle that load efficiently. The existing body of research has provided tantalizing hints of that potential, but little hard evidence.

Personalization is a fundamental advantage of machine-learning-based spam classification algorithms. Everyone receives different kinds of mail, both legitimate and spam. Intuitively, a filter finely tuned to the peculiarities of the kinds of mail you receive is more likely to work better for you than a generic filter shared with thousands of other people. Furthermore, if everyone has a different filter, it will be much harder for spammers to craft a message that systematically circumvents the system. Therefore, an enterprise spam filter based on machine-learning-based spam classification algorithms must be able to efficiently provide personalized filtering on the server. The question is: Can this be done? Machine learning algorithms are notoriously slow, and the existing spam classification research has provided little direct evidence to the contrary. Section 3.2 will propose an architecture for machine-learning-based enterprise spam filters that appears to be efficient and scalable.

Assuming that these algorithms can be used to build efficient enterprise spam filters, the next question is: Will they work? That is, will they work for *everyone*? There needs to be research performed that systematically evaluates the effectiveness and safety of spam classifiers on large numbers of complete, real email streams, representing a variety of different kinds of users. To date, the privacy barrier has prevented researchers from getting access to that kind of test data. Section 3.3 will introduce a novel experimental methodology - one that will make it possible to evaluate these algorithms in a realistic way for the first time.

Another advantage of many machine-learning-based spam classification algorithms is the ability to tune decision thresholds, enabling administrators to manage risk by making trade-offs between effectiveness and safety. The question is: Can these algorithms be both effective *and* safe?

In summary, this report will address the following questions:

- Is there an efficient, scalable architecture for an enterprise spam filter that uses spam classification algorithms to implement personalized filters on shared servers?
- Is there a way to evaluate spam classification algorithms on large numbers of individual, real, complete email streams?
- How effective are some of the more popular algorithms when evaluated on this more realistic data?
- How effective are they when tuned for varying degrees of safety?
- How many training messages do they need to see before they become effective?
- How long are algorithm execution times?
- How much data must be stored on the server for each user?

3.2 A Scalable Architecture For Enterprise Spam Filters

The key to building a scalable enterprise spam filter using machine-learning-based spam classification algorithms is recognizing that most of the hard work can still be done on the client side. In general, it is training the classifiers that takes time; classification is very fast for many algorithms.

To make this work, we need to borrow an idea from cryptography - a small key that “plugs” into the algorithm to make it work. In this case, a personalized “filter key” is generated on the client by the classifier’s training algorithm. The filter key is then passed to the server, where it is stored in a database. Whenever a message arrives, the server retrieves the filter key for the appropriate user, and plugs it into the classifier to do the filtering. Figure 3.1 illustrates this idea.

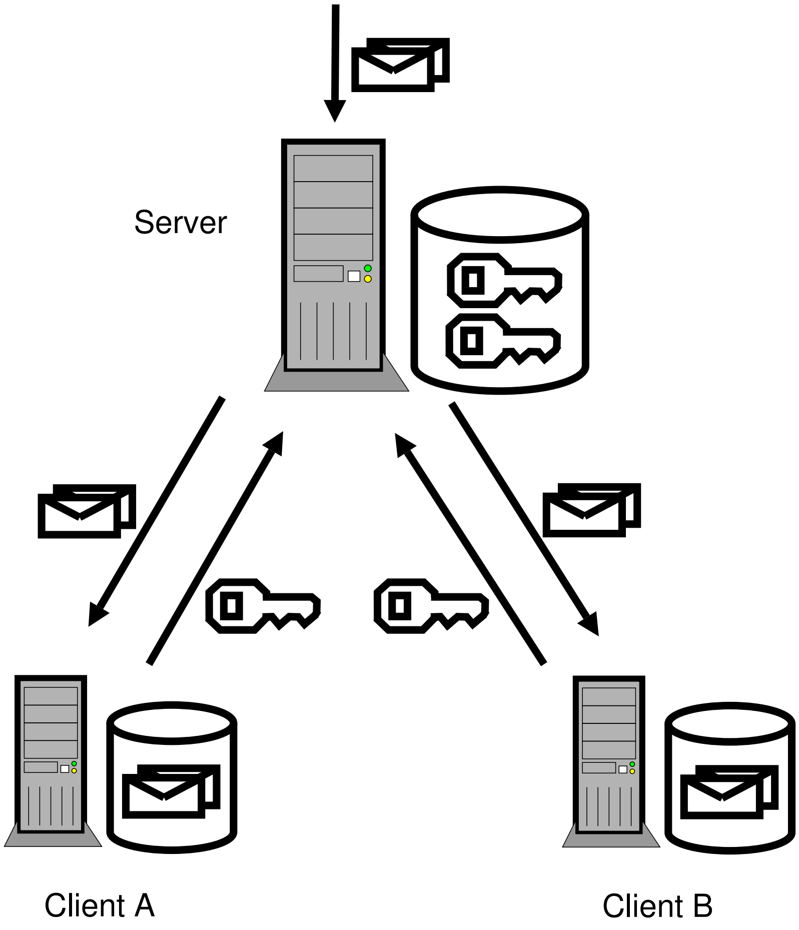

Figure 3.1: Proposed architecture for an enterprise spam filter.

If you can assume that classification time is “very fast”, training time is “reasonable”, and the filter key is “small”, then what you have left is basically an engineering exercise. After looking at some implementation considerations, we’ll revisit those assumptions.

A *distributed* enterprise spam filter will have client software (perhaps a plug-in for a popular email client like Microsoft Outlook) integrated with the server at the edge of the network. On the client side, the user will have a prominent “mark as spam” button seamlessly integrated into the graphical user interface of the email client. The client software will maintain a local database of all the user’s most recent messages, including the classification of each. Periodically, during computer idle time, the software will invoke the spam classification algorithm to train (and test) itself on the database of messages. Once it has a large enough database, and the self-tests are successful enough, the client automatically generates a compact filter key and connects to the server.

Upon connection, the client will upload the filter key to the server. The server software will store this key in a database, indexed by user ID. As each incoming email message arrives, the server will determine the user ID, retrieve that user's filter key from the database (or a cache), plug the key into the spam classification algorithm, and classify the message. If the message is determined to be legitimate, it is delivered as usual. If not, the sender, subject line, and perhaps the first few lines of the message are appended to the user's "spam digest" stored somewhere on the network. The spam message is then discarded.

Once a day, or whenever the user wants to, the user's email client software will move the compact spam digest to the local machine. Then, it can be checked for false positives at the user's convenience. If a false positive is suspected, the user has the sender's address to ask them to resend the message (and it will be accepted the second time, as explained below).

Periodically (perhaps once a week) the client software will produce an updated filter key, based on the most recent messages received (including the few spam messages that slipped through). In this way the filter smoothly adapts to the latest tactics used by spammers.

The filter key will consist of the following:

- The dictionary generated by the feature selection process (see Section 2.4);
- The internal data from the trained classifier, representing what it has "learned";
- A "white list" maintained automatically by the client software, containing every address that the user has sent a message to, along with manual entries added by the user.

The server uses the filter key as follows: first, a message from anyone on the white list automatically bypasses the filter; second, the dictionary is used to map the incoming message to an instance vector(see Section 2.5); and third, the internal classifier data is simply loaded into place in the server's classifier, so it can classify the instance vector.

The client software will provide a mechanism for the user to extract an address from the spam digest when a false positive is found. This can be used to request the sender to resend the message that got blocked. To prevent the message from being blocked a second time, that address would be automatically added to the white list, and the resulting new version of the filter key would be sent to the server. This means that spam can be blocked from entering the network, but no legitimate messages need ever be lost.

In summary, if our assumptions hold, this architecture will meet the “automatic”, “adaptive” and “efficient” requirements for an enterprise spam filter. The remaining work is to back up our assumptions, and show that there are “effective” and “safe” algorithms available. Those algorithms will be described below, following the proposal of a new methodology for evaluating their effectiveness and safety.

Data to support the efficiency assumptions will be presented in Section 5.7.

3.3 A Better Way To Evaluate Spam Classifiers

To accurately evaluate spam classification algorithms, they must be tested on real data. It’s not that we need *larger* datasets; it’s that we need *more* of them! And they should be complete, not just the messages that people saved. Also, mail from different people should not be mixed. This is the kind of data that enterprise spam filters will deal with in real life, so we need to find out how well they will handle it.

In order to overcome the privacy barrier, this thesis introduces a novel experimental methodology that makes it possible to systematically evaluate the algorithms on complete, real email streams from many individuals, without violating their privacy. The concept is a distributed, blind test infrastructure.

Similar to the architecture described above, we let the clients do most of the work. Test volunteers install a piece of software that integrates with their email clients. That software superficially operates as a client-side spam filter. But the corpus of labeled emails it builds locally for training purposes also serves as a private test bed for experiments on different algorithms. During idle periods, the software runs experiments on the growing corpus, and reports (just) the results back to a central server. The server aggregates the test results from all the volunteers. The client software can also download new experiments from the server as researchers release them.

This design makes it possible to gather test results from a large variety of people. Not only is their privacy protected, but they get the benefit of a state-of-the-art spam classifier.

There are, however, some disadvantages. Since researchers can’t see the data, they can't easily make sure the software is being used properly. And it is also not easy to analyze what types of messages the classifiers have trouble with. However, with a carefully designed user interface, and thorough logging and reporting, much of the disadvantage can be minimized.

It is very important to design the user interface with the primary goal in mind: building a complete database of *correctly* labelled messages. Therefore, the design is not necessarily going to be the same as that of a commercial spam filter product. Here we need to try to ensure that every single incoming message gets labelled correctly, so that we can run experiments on complete sets of email. Commercial products may be designed to let users control which messages are used for training.

This is the concept behind the experimental methodology used for this thesis. The implementation details will be explained in Chapter 4.

3.4 Three Promising Algorithms

This report evaluates three of the most popular algorithms in the spam classification literature: Naive Bayes, Support Vector machines, and AdaBoost (boosted decision trees). All three have shown very promising results, appear to be relatively efficient, and produce confidence-weighted classifications suitable for tuning with thresholds. Each will be introduced briefly below, with references to more detailed descriptions.

3.4.1 Naive Bayes

The Naive Bayes classifier is a simple statistical algorithm with a long history of providing surprisingly accurate results. It has been used in several spam classification studies [16, 13, 1, 2, 6, 9, 8, 20], and has become somewhat of a benchmark. It gets its name from being based on Bayes's rule of conditional probability, combined with the "naive" assumption that all conditional probabilities are independent (given the class) [22].

During training, the Naive Bayes classifier examines all of the instance vectors from both classes. It calculates the prior class probabilities as the proportion of all instances that are spam ($Pr[spam]$), and not-spam ($Pr[notspam]$). Then (assuming binary attributes) it estimates four conditional probabilities for each attribute: $Pr[true|spam]$, $Pr[false|spam]$, $Pr[true|notspam]$, and $Pr[false|notspam]$. These estimates are calculated based on the proportion of instances of the matching class that have the matching value for that attribute.

To classify an instance of unknown class, the "naive" version of Bayes's rule is used to estimate first the probability of the instance belonging to the spam class, and then the probability of it belonging to the not-spam class. Then it normalizes the first to the sum of both to produce a spam confidence score between 0.0 and 1.0. Note that the denominator of Bayes's rule can be omitted because it is cancelled out in the normalization step. In terms of implementation, the numerator tends to get quite small as the number of attributes grows, because so many tiny probabilities are being multiplied with each other. This can become a problem for finite precision floating point numbers. The solution is to convert all probabilities to logs, and perform addition instead of multiplication. Note also that conditional probabilities of zero must be avoided; instead a "Laplace estimator" (a very small probability) is used.

It is important to note that using binary attributes in the instance vectors makes this algorithm both simpler and more efficient. Also, given the prevalence of sparse instance vectors in text classification problems like this one, binary attributes offer the opportunity to implement very significant performance optimizations. Figure 3.2 presents the Naive Bayes training and classification algorithms used.

```
Naive Bayes Training Algorithm:
$priorProbSpam$ = proportion of training set that is spam
$priorProbNotSpam$ = proportion of training set that is not-spam
For each attribute $i$:
    $probTrueSpam[i]$ = prop. of spams with attribute $i$ true
    $probFalseSpam[i]$ = prop. of spams with attribute $i$ false
    $probTrueNotSpam[i]$ = prop. of not-spams with attribute $i$ true
    $probFalseNotSpam[i]$ = prop. of not-spams with attribute $i$ false

Naive Bayes Classification Algorithm:
$probSpam = priorProbSpam$
$probNotSpam = priorProbNotSpam$
For each attribute $i$:
    if value of attribute $i$ for message to be classified is true:
        $probSpam = probSpam \times probTrueSpam[i]$
        $probNotSpam = probNotSpam \times probTrueNotSpam[i]$
    else:
        $probSpam = probSpam \times probFalseSpam[i]$
        $probNotSpam = probNotSpam \times probFalseNotSpam[i]$
$spamminess = probSpam/(probSpam + probNotSpam)$
```

Figure 3.2: Naive Bayes training and classification algorithms.

3.4.2 Support Vector Machines

Support vector machines (SVMs) are relatively new techniques that have rapidly gained popularity because of the excellent results they have achieved in a wide variety of machine learning problems, and because they have solid theoretical underpinnings in statistical learning theory [5]. They have also been used in several spam classification studies [7, 10, 8, 11].

SVMs have an intuitive geometrical interpretation [22]. Essentially, during training, the algorithm attempts to find a maximum-margin hyperplane that separates the instance vectors of the two classes. This hyperplane can be completely defined by just the instances, called support vectors, from each class that are closest to it. This makes SVMs highly resistant to overfitting. If the two classes are not linearly separable, nonlinear SVMs can be used to find nonlinear class boundaries; however, the much faster linear SVMs have achieved excellent results in spam classification.

There is an efficient algorithm for training SVMs, called Sequential Minimal Optimization (SMO) [14]. Linear SVMs are also extremely fast classifiers - classification amounts mainly to computing the dot product of the instance vector with a weight vector produced during training. Also, similar to Naive Bayes, binary attributes speed up SVMs, and the sparse instance vectors offer substantial performance optimization opportunities.

SVMs produce confidence weighted classification scores that, using a sigmoid function, can be mapped to a score between 0.0 and 1.0. Figure 3.3 presents the linear SVM training and classification algorithms used.

```
Linear SVM Training Algorithm:
Use SMO to calculate $b$ and $w[i]$ for each attribute $i$

Linear SVM Classification Algorithm:
$output$ = (dot product of $w$ and instance vector to be classified) - $b$
$spamminess = 1.0/(1.0 + exp(-output))$
```

Figure 3.3: Linear SVM training and classification algorithms.

3.4.3 AdaBoost

AdaBoost is another relatively new, but popular algorithm. It has also been used with great success in spam classification [3]. It is one of a family of “boosting” algorithms; these techniques attempt to “boost” the accuracy of a “weak” classifier [18]. The algorithm proceeds in a series of rounds, each time training a different version of the weak classifier. The difference is based on a weight value for each training instance; after each round, all the weights are updated in a way that emphasizes the instances that previous versions of the weak classifier had trouble with.

For this thesis, the AdaBoost.MH variant [19] is used, because it supports weak classifiers that produce confidence-weighted decisions. It requires that the weak classifier output a positive score to indicate spam, and a negative score to indicate not-spam. The magnitude of the score indicates the confidence. We used decision tree "stumps" (single-level trees) for the weak classifier. While multi-level trees have shown better results [3], this thesis is highly focused on efficiency, and stumps are faster.

During classification, the algorithm simply tallies up the spam and not-spam votes from all the weak classifiers, and produces a normalized spam confidence score between 0.0 and 1.0.

Figure 3.4 presents the AdaBoost.MH training and classification algorithms used.

```
AdaBoost.MH Training Algorithm:
Initialize weights on training instances
For each round $i$:
    Train a new weak classifier $i$ on weighted training instances
    Use accuracy of weak classifier $i$ to update weights on training instances

AdaBoost.MH Classification Algorithm:
For each round $i$:
    $weakScore$ = spamminess score of message calculated by weak classifier $i$
    if $weakScore$ is positive:
        $spamScore = spamScore + weakScore$
    else:
        $notSpamScore = notSpamScore + (weakScore \times -1.0)$
$spamminess = spamScore/(spamScore + notSpamScore)$
```

Figure 3.4: AdaBoost.MH training and classification algorithms.

Chapter 4

Implementation and Experimental Design

This chapter describes the extensible C++ API built to support the research, and how the experiments were designed and carried out.

4.1 An Extensible C++ API for Spam Classification Research

To support this thesis, we designed and built an extensible, object-oriented application programming interface (API) for spam classification research. This is not a general machine learning library; it is specifically designed for building and testing spam classifiers. Some of the top-level classes include:

- `Document` - represents an RFC 822 compliant email message (RFC 822 is the basic format specification for Internet email messages);
- `Instance` - represents an instance vector mapped from a `Document`;
- `Corpus` - represents a collection of `Documents`;
- `Dictionary` - stores the tokens represented by each attribute of an `Instance`;
- `Parser` - parses a `Document` into a list of tokens;
- `FeatureSelector` - given a set of token lists, produces a `Dictionary`;
- `Vectorizer` - given a list of tokens and a `Dictionary`, produces an `Instance`;
- `Trainer` - given a `Corpus`, produces a trained `Classifier`;
- `Classifier` - given an `Instance`, returns a "spamminess" score between 0.0 and 1.0;
- `Evaluator` - given a set of `Documents`, and a set of test configurations, runs a series of cross validation experiments.

The API is designed for maximum flexibility in trying out different techniques. Using the object-oriented technique called *inheritance*, a wide variety of subclasses can be plugged in and tested. For example, the API already contains a subclass of `Instance` called `BinaryInstance`; it would be easy to add another called `TFInstance` to implement term frequency instance vectors. Continuing with that example, just as the subclass `BinaryVectorizer` maps the `Parser` output to a `BinaryInstance`, a `TFVectorizer` could produce `TFInstances`. Because of inheritance, all these subclasses can still be submitted for tests in the `Evaluator`, because the `Evaluator` only deals with top-level classes. Similarly, a wide variety of parsing, feature selection and classification algorithms can be easily added to the API.

Although Java would have been (arguably) a more portable language to use, implementation efficiency is a critical requirement for this research. Also, C++ allows more direct integration with popular Windows and Unix email clients.

4.2 The SpamTest Platform

We built and deployed a test platform that implements the new experimental methodology introduced in Section 3.3. Originally, we hoped to build a fully distributed system, using software plug-ins for one or more of the most popular email clients. Unfortunately the number of different email clients used by our group of volunteer testers was almost as big as the number of volunteers themselves! Limited time and resources forced us to take a different approach. We decided on a design that would work for any email client - a design that involved running our client software on the Dalhousie University Faculty of Computer Science mail server! Needless to say, efficiency suddenly became even more critical.

The “SpamTest” platform takes advantage of a popular Unix mail processing utility called “procmail”. When an email arrives for a SpamTest volunteer, procmail is invoked by the mail server. Procmail runs the email through our trained classifier program called “spamflagger”, which inserts a label into the beginning of the message’s subject line. That label loudly declares the message to be either “SPAM” or “OK” and includes a “spamminess” score and a message ID. When the volunteer sees a message with an incorrect label, she forwards the message to herself. Procmail intercepts this “correction message” and passes it to a program called “spamcorrector”. This is the extent of the SpamTest experience for users.

Behind the scenes though, much more is going on. When spamflagger processes each message, it always assumes it is right. Therefore, it makes a copy of the message, and stores it in either the spam or not-spam side of the volunteer's private message database. When spamcorrector receives a correction message, it examines the subject line label, extracts spamflagger's original classification and the message ID, and moves the specified message to the opposite side of the database.

At this point, it is worth acknowledging that this may not be the best way to design the user interface of a real spam filter product. However, our primary goal for the user interface is *to maximize the accuracy of the message classifications in the database, with as little user effort as possible.* Of secondary importance is providing something of value to our volunteers - a very accurate spam flagger, that makes sorting through their inbox less painful. The labels also have a special feature. If you sort your inbox alphabetically by subject (easy on many email clients), the "least spammy" messages float to the top, and the "most spammy" messages sink to the bottom.

To protect the privacy of our volunteers, we use the file access control mechanisms provided by Unix. The private message databases accumulated by SpamTest for each volunteer can only be accessed by that volunteer, or by a program invoked by her. Therefore, it is a bit of a challenge to arrange for periodic or infrequent training and testing. However, programs invoked by procmail on behalf of a volunteer are treated by Unix as if they were invoked by her. So procmail helps us overcome this problem too. Procmail is configured to recognize and intercept special "train" and "run tests" commands from a particular email address. When the "train" command is received (typically at night), procmail invokes a program called "spamtrainer" that trains the volunteer's classifier on the messages in her private database. Similarly, a "run test" command triggers procmail to invoke the "spamtester" program, which runs a set of experiments on the private database, and makes just the results accessible to researchers.

There were some interesting implementation issues. First of all, the Dalhousie Faculty of Computer Science already runs a popular spam filter called "Spam Assassin" on all incoming messages. This filter adds custom headers to each message, indicating (among other things) whether the message is spam or not. We used the Spam Assassin tags to generate an initial training set of reasonable quality to bootstrap our classifiers. The headers added by Spam Assassin were stripped out before the messages were used to train the classifiers, but their spam/no-spam tags were used as the class labels of the messages in the initial training set. eem: the following text was taken out and replaced by the previous text, to make the use of Spam Assassin tags more clear: "We took advantage of this to "bootstrap" our classifiers, so volunteers would not have a huge number of corrections to make when first starting out. Note that the headers added by Spam Assassin were stripped out before the messages were used to train the classifiers!" Second, we also attempted a "blind" database cleanup just before the final experiments were run. We again took advantage of the Spam Assassin headers - if a message was flagged as spam by SpamAssassin, yet the user had it marked as not-spam, we used SpamTest to auto-generate a request for them to double-check that message. The opposite was not possible, because there were far too many spam messages missed by Spam Assassin. We did not want to flood our volunteers with messages to double-check! We did however have SpamTest generate a few double-checks in extreme cases of messages that all of the classifiers had trouble with. As a whole, these double-checks resulted in quite a few corrections, so it is fair to say noise in the data is a significant challenge under this methodology. After the cleanup, experimental results improved noticeably for all the classifiers.

4.3 Experiments

The `Evaluator` class, described in Section 4.1, makes it easy to "mix-and-match" different `Parsers`, `FeatureSelectors`, `Vectorizers` and `Classifiers` in experiments. One of each is specified in a single test configuration, and any number of test configurations can be added to the Evaluator before the experiment runs. Each configuration is trained and tested on the same sequence of messages in a ten-fold cross-validation (with stratification) procedure.

For the SpamTest platform, described in Section 4.2, the "spamtester" program assembles the test configurations and starts the `Evaluator`. Different configurations were run on different nights.

The first experiments were used to determine the best configuration for each classifier ("best" sometimes meaning a compromise between effectiveness and efficiency). Then we ran the best configurations for each classifier against each other. The experiments are listed below. Note that each volunteer's database was locked before running these final experiments, meaning no messages were added, deleted or changed between experiments.

4.3.1 Feature Extraction

For each of the classifier algorithms we tried out a series of different parsing techniques. Our baseline `Parser` included the values of all header fields, as well as the entire body text (although the `Document` class strips out MIME attachments before parsing, leaving only the text and HTML portions of a MIME-formatted message, along with the MIME headers for each attachment). In all cases, the characters `",.!?-:<>*;()[]’‘` as well as the space, tab and newline characters were used as token delimiters. Carriage returns were stripped out by the Document class before parsing. In future studies, different tokenization strategies may well be worth evaluating. Note that tokens were truncated to a maximum length of 30 characters.

We tried the baseline parser plus four different variations of it for each classifier. Those variations are:

- Including header names as tokens (not just header values);
- Using “header tags” (prepending each header token with the name of the header it came from);
- Omitting headers altogether (except the subject line); and,
- Omitting the body text.

4.3.2 Feature Selection

Related to the parser experiments, we also tried out each classifier with a special `FeatureSelector` that converted all tokens to lower case before building the dictionary, and compared that to the baseline configuration.

We did not incorporate stemming or stoplists into our evaluations, because the results from previous studies [13, 7, 2, 1] did not convince us that the benefits would justify the efficiency penalties.

Our major feature selection experiments consisted of using mutual information (MI) [16, 2, 1, 8, 20] to determine the best number of attributes to work with for each classifier. We tried out each classifier using 50, 100, 250, 500, 1000, 2500, 5000 and 10000 attributes. Note that other authors [7] have recommended letting the SVM and decision tree algorithms perform their own feature selection. Since it was convenient, we decided to try using MI anyway.

4.3.3 Classification Algorithms

As explained in Section 3.4, this thesis evaluates the Naive Bayes, linear SVM and AdaBoost.MH (with decision tree stumps) algorithms. Before testing the algorithms against each other, we first determined the number of rounds (see Section 3.4.3) needed by AdaBoost.

Then, based on the previous experiments, we ran the chosen baseline configurations of the three algorithms against each other. We repeated this experiment on a variety of corpus sizes: 50, 100, 200, 400 and 800 messages. Each time, we simply used the most recent messages in each database, regardless of class. For example, the last test ran on the 800 most recent messages from each volunteer's private database (for those who had large enough databases).

4.4 Evaluation Plan

We collect a variety of data from each experiment. For each volunteer, one experiment runs each night, yielding one test report. The test report produced by the `Evaluator` contains the following information:

- The username and test time;
- Total number of spams and not-spams in the corpus;
- Detailed descriptions of each test configuration used in the experiment;
- Information about each fold (size and class distribution of training set and test set);
- Classifier outputs (for each test configuration) for each message; and
- Execution times (for each test configuration) for feature selection, parsing, instance mapping, training and classification.

A "test report reader" program analyzes these test reports and produces summaries, as well as metrics for each test configuration. From a volunteer's test report, we calculate the spam recall for each test configuration at the decision threshold that fixes the false positive rate at 5, 2, 1, 0.5 and 0 percent. Note that linear interpolation is used to estimate spam recall values for those precise false positive rates.

We compare test configurations by averaging these metrics across all volunteers.

Chapter 5

Results and Analysis

This chapter discusses the results of our experiments. As described in Section 4.2, we had to limit our study to only the volunteers who have email accounts with the Faculty of Computer Science. In the end, this reduced us to seven people who received a large enough volume of mail over the duration of our study. Those seven people consisted of three professors, two graduate students and two administrative staff members. Although this is a much smaller number than we had hoped for, we believe it is enough to test our new methodology. It also represents a reasonable diversity of user types.

All results presented below represent the average across all seven users, unless otherwise noted. Where fixed corpus sizes are referred to, the most recent messages in each user's database were selected.

Complete results for all experiments are listed in Appendix A.

5.1 Baseline Configurations

The following are the baseline test configurations, used in all experiments except where otherwise noted. These were chosen on the basis of the results of the initial experiments, described in Sections 5.2, 5.3 and 5.4.

- `Parser`: header names omitted, header tags not used, header values included, body text included.
- `FeatureSelector`: case-sensitive, best 1000 attributes selected using mutual information.
- `AdaBoostMHClassifier`: 50 rounds.
- Corpus size: maximum 500 most recent messages (one user had only 401 messages in his database).

5.2 Feature Selection

Table 5.1 shows the results of the main feature selection experiments, which tested the classifiers versus the number of attributes. These experiments used the baseline configurations, except the AdaBoost classifier used 100 rounds. These results show no significant justification for using more than 1000 attributes, so that was chosen for the baseline configuration for each classifier in subsequent experiments.

Number of	Spam Recall(%) @ 1% FPR		
Attributes	Naive Bayes	SVM	AdaBoost
50	75.5	74.6	82.9
100	86.1	93.5	93.5
250	93.1	93.7	95.2
500	96.4	95.9	96.9
1000	97.5	96.8	97.0
2500	97.7	97.2	96.8
5000	97.9	97.1	97.1
10000	98.2	97.3	96.8

Table 5.1: Spam recall at 1% false positive rate versus number of attributes.

We also compared the baseline configurations against versions that used a case-insensitive feature selector. The results in Table 5.2 show no significant benefit to converting all tokens to lower case before constructing the dictionary.

	Spam Recall(%) @ 1% FPR		
Configuration	Naive Bayes	SVM	AdaBoost
Baseline	97.8	97.2	95.7
Case-Insensitive	96.7	96.9	95.6

Table 5.2: Spam recall at 1% false positive rate versus case sensitivity.

5.3 AdaBoost Rounds

We tested the baseline AdaBoost configuration against the number of rounds (see Section 3.4.3). The results, shown in Table 5.3, provide no justification for using more than 50 rounds, so that was chosen for the baseline AdaBoost configuration in subsequent experiments.

Number of Rounds	Spam Recall(%) @ 1% FPR
25	93.5
50	94.9
100	94.6
200	93.9
400	95.0

Table 5.3: AdaBoost spam recall at 1% false positive rate versus number of rounds.

5.4 Feature Extraction

We tested the baseline configuration of each classifier against several different parsing strategies (see Section 4.3.1). Table 5.4 compares the baseline against a parser that includes header names. Table 5.5 compares the baseline against a parser that includes "header tags". Table 5.6 compares the baseline against a parser that omits the headers altogether (except for the subject). Table 5.7 compares the baseline against a parser that omits the message body.

These results suggest that there is little value in including header names or adding "header tags"; however, they also clearly show that both the non-subject header values and the body text are valuable for classification. We did not test omitting the subject line itself, but it is presumed to be valuable as well.

	Spam Recall(%) @ 1% FPR		
Configuration	Naive Bayes	SVM	AdaBoost
Baseline	97.5	96.6	94.2
Incl. Header Names	97.8	97.1	95.0

Table 5.4: Spam recall at 1% false positive rate versus inclusion of header names.

	Spam Recall(%) @ 1% FPR		
Configuration	Naive Bayes	SVM	AdaBoost
Baseline	97.7	97.3	97.1
Use Header Tags	96.6	95.9	95.0

Table 5.5: Spam recall at 1% false positive rate versus use of "header tags".

Configuration	Spam Recall(%) @ 1% FPR		
	Naive Bayes	SVM	AdaBoost
Baseline	97.6	97.5	95.9
Omit Headers	95.4	92.2	93.1

Table 5.6: Spam recall at 1% false positive rate versus inclusion of header values.

Configuration	Spam Recall(%) @ 1% FPR		
	Naive Bayes	SVM	AdaBoost
Baseline	97.7	97.7	96.4
Omit Body	84.0	88.7	89.1

Table 5.7: Spam recall at 1% false positive rate versus inclusion of message body.

5.5 Classification Algorithms versus Corpus Size

With the baseline configurations finalized for all three algorithms, we tested them against each other at several different corpus sizes. Figure 5.1 shows the average proportion of spam in users' corpora for each size. Note that one user was omitted from these experiments, because he did not have enough messages in his database to participate in the 800-message test. Table 5.8 and Figure 5.2 show the results.

Four interesting observations can be made:

1. There appears to be little performance difference between Naive Bayes and SVMs.

2. Both Naive Bayes and SVMs appear capable of excellent performance on corpora as small as 50 messages.

3. The AdaBoost algorithm (using just decision tree stumps) appears to be much more reliant on larger corpus size.

4. All three algorithms appear to be approaching peak performance at 400 messages.

5.6 Classification Algorithms versus False Positive Rate

The three algorithms were also compared on the basis of spam recall at different false positive rates. Table 5.9 and Figure 5.3 show the results for a corpus size of 400 messages. A statistical significance analysis on the results of this experiment is

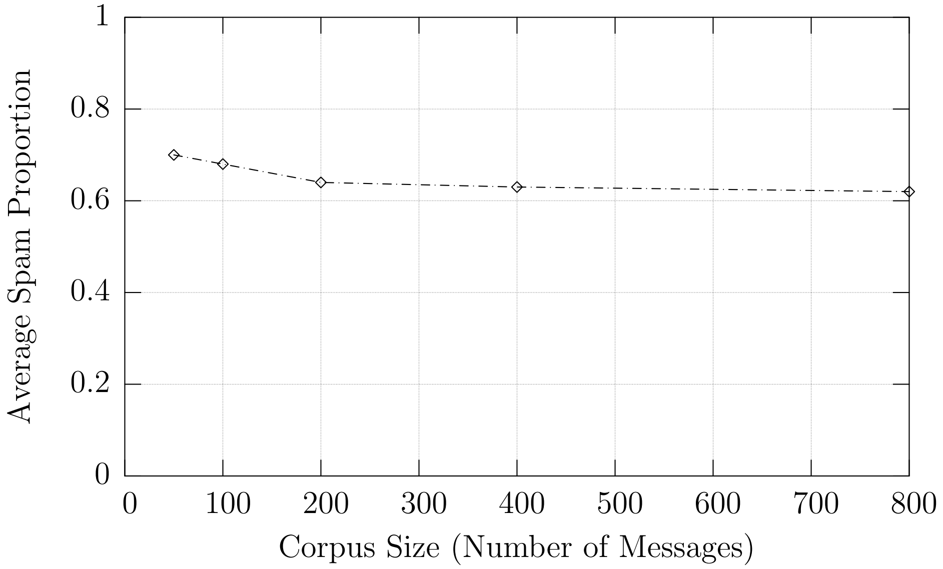

Figure 5.1: Average spam proportion versus corpus size.

Number of	Spam Recall(%) @ 1% FPR		
Messages	Naive Bayes	SVM	AdaBoost
50	91.9	92.9	38.5
100	94.8	93.7	63.9
200	93.3	93.1	83.4
400	96.8	96.0	93.5
800	95.9	96.9	95.1

Table 5.8: Spam recall at 1% false positive rate versus corpus size.

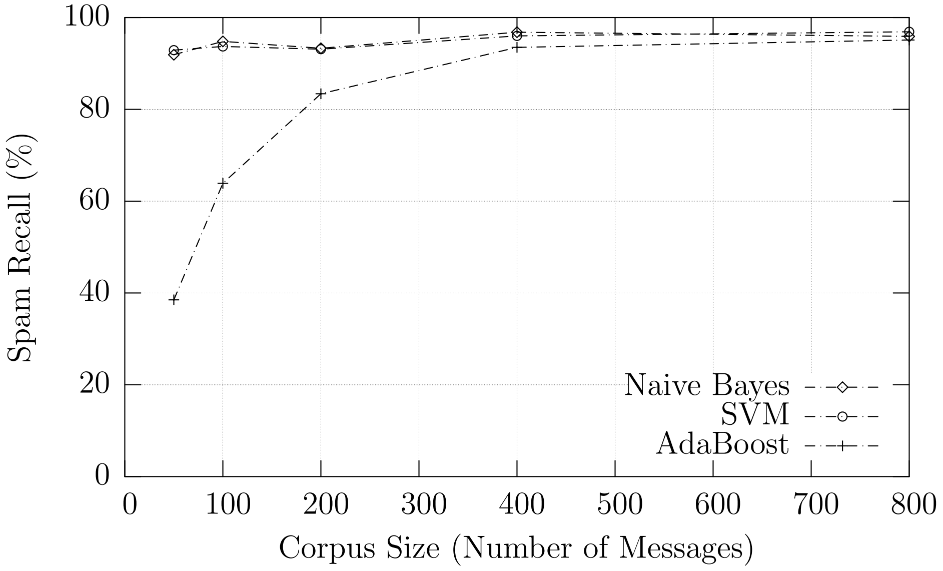

Figure 5.2: Spam recall at 1% false positive rate versus corpus size.

presented in Appendix B. None of the differences between means is significant at the 0.05 level.

Five interesting observations can be made:

1. Again we see little measurable performance difference between Naive Bayes and SVMs.

2. AdaBoost (using just decision tree stumps) appears to be slightly less effective, particularly at high "safety" levels (although this difference is not statistically significant).

3. All three appear to be very effective even when tuned for zero false positives.

4. All three appear to offer a well-behaved trade-off between effectiveness and safety, that can be tuned using the decision threshold.

5. For all three algorithms, roughly 90% of spam scored higher on the "spamminess" scale than the *worst* false positive.

The last observation has important implications for the safety of machine-learning-based spam classification algorithms. It appears that the false positives tend to be among the *least confident* classifications. This information may be used to help safeguard against the loss of false positives. .

	Spam Recall(%)		
FPR(%)	Naive Bayes	SVM	AdaBoost
5.0	98.9	99.5	98.7
2.0	98.0	97.8	97.4
1.0	97.2	96.5	94.4
0.5	95.9	95.6	91.9
0.0	94.1	93.8	89.5

Table 5.9: Spam recall versus false positive rate.

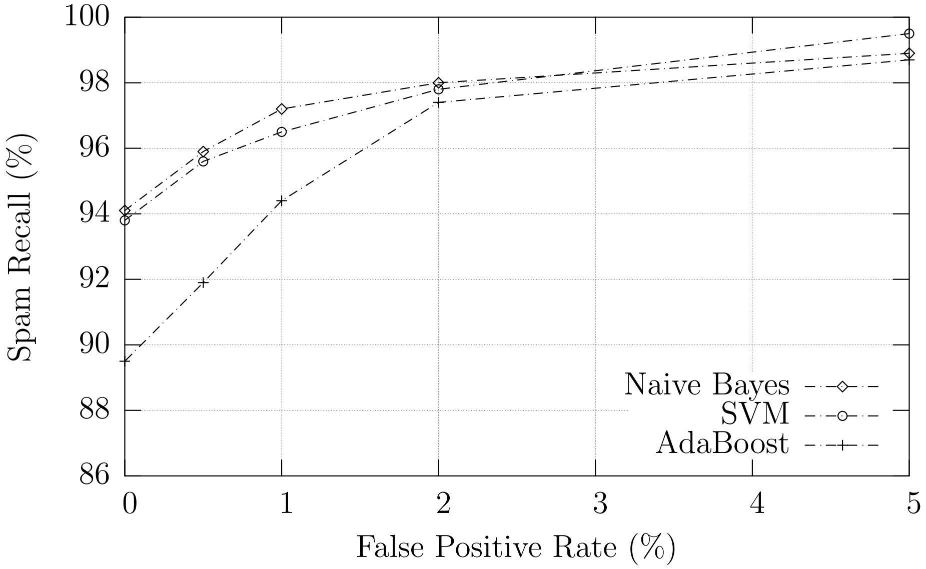

Figure 5.3: Spam recall versus false positive rate.

5.7 Classification Algorithm Efficiency

We also measured execution times for various tasks. Table 5.10 shows the results. These measurements are for a 400-message corpus. Note that the "feature selection", as well as "parsing and instance mapping" times are the same for all three algorithms, because they all shared the same baseline configuration for those tasks. Also note that execution times for one user were omitted as an outlier, believed to be due to a time of day anomaly on a shared mail server.

Table 5.11 shows the estimated average total training and classification times, under the assumptions of 1000 attributes, 400 messages and 50 AdaBoost rounds. Note that total training time includes parsing, feature selection, instance mapping and training on 400 messages, while total classification time includes parsing, instance mapping and classification for one message. This table shows that while the differences between algorithm training and classification times are extreme, overall they are small compared to the message preprocessing tasks that are the same regardless of algorithm. Note that the estimated total times do not consider disk I/O.

	Execution Time (ms/message)		
Task	Naive Bayes	SVM	AdaBoost
Feature Selection	197	197	197
Parsing and Instance Mapping	35.3	35.3	35.3
Training	0.20	62.3	20.7
Classification	0.34	0.03	0.02

Table 5.10: Algorithm task execution times (per message).

	Total Execution Time		
Task	Naive Bayes	SVM	AdaBoost
Training (min)	1.6	2.0	1.7
Classification (ms)	35.6	35.3	35.3

Table 5.11: Algorithm estimated average total execution times.

We also measured the average size of the "filter key" produced by each algorithm. These are shown in Table 5.12, both with and without the dictionary portion of the key (see Section 3.2) for ease of comparison. eem: can the key be used by the mail server for classification without the dictionary portion? It seems that you must include the dictionary portion, since every training session extracts features again. These measurements assume 1000 attributes and 50 AdaBoost rounds. Note that the Naive Bayes, SVM and dictionary key sizes are all roughly proportional to the number of attributes, while the AdaBoost key size is roughly proportional to the number of rounds (because AdaBoost stores data for one weak classifier per round). Also note that the average dictionary key size was 6742 bytes, and is the same for all three algorithms.

	Average Key Size (bytes)		
Dictionary Incl?	Naive Bayes	SVM	AdaBoost
Without Dictionary	33652	9211	1071
With Dictionary	40394	15953	7813

Table 5.12: Algorithm average filter key size.

Chapter 6

Conclusions and Future Work

The goal of this thesis is to make a convincing argument that a practical enterprise spam filter can be built using machine learning algorithms.

We have defined the critical requirements for enterprise spam filters, and the information that is still needed to determine if machine-learning-based spam classification algorithms can be used to meet those requirements. Now we will revisit those unanswered questions and examine the evidence provided by our experiments.

We proposed that, in order to protect networks from the escalating costs of spam, an enterprise spam filter must be automatic, adaptive, effective, safe and efficient. We also stated that while machine-learning-based spam classification algorithms are known to satisfy some of these requirements, the following questions still need to be answered:

- Is there an efficient, scalable architecture for an enterprise spam filter that uses spam classification algorithms to implement personalized filters on shared servers?
- Is there a way to evaluate spam classification algorithms on large numbers of individual, real, complete email streams?
- How effective are some of the more popular algorithms when evaluated on this more realistic data?
- How effective are they when tuned for varying degrees of safety?
- How many training messages do they need to see before they become effective?
- How long are algorithm execution times?
- How much data must be stored on the server for each user?

The following sections will attempt to answer those questions on the basis of our experimental results.

6.1 An Efficient Architecture

In Section 3.2, we presented an architecture for an enterprise spam filter; its distributed design makes it seem possible that personalized filtering on shared servers can be both efficient and scalable. This design relies on three assumptions that we tested in this study. First, we assumed that classification (which must run on the server) is “very fast”. Second, we assumed that training (which runs during idle periods on client machines) time is “reasonable”. Third, we assumed that the “filter key” (produced by training and passed to the server to use for classification) is “small”. Our results support those assumptions.

Based on our measurements, the estimated average total classification time for a single incoming message arriving at the Faculty of Computer Science mail server is about 35 milliseconds for all three algorithms. The vast majority of that time is spent parsing the message and mapping it to an instance vector. It is hard to imagine *any* spam filtering solution that will not have to parse and/or process each incoming message in some way; therefore using machine-learning-based spam classification algorithms for spam filtering is probably at least *no worse* than any other approach in terms of processing load on the server.

In terms of training times, our estimated average total training time (for 400 messages) was roughly two minutes for all three algorithms, again dominated by message preprocessing tasks. It would be difficult to argue that performing two minutes of work per user, once a week during idle periods is “unreasonable”. Our tests actually ran entirely on a working mail server; it is difficult to predict if training would typically be significantly faster or slower on a client machine.

Finally, our results show the largest average filter key size to be roughly 40 kB. Adding in a white list averaging the same size as the dictionary increases that to about 47 kB. To put that in perspective, all the keys for 1000 users would total up to about 45 MB - which could easily fit entirely in memory for a mail server of even modest size. eem: why is the white list on the average 7kB? In summary, our experiments have provided strong evidence that our proposed architecture would be efficient and scalable for any of the three algorithms tested.

6.2 A Better Way To Evaluate Spam Classifiers

In Section 3.3, we presented a novel experimental methodology that overcomes the privacy barrier, finally making it possible to evaluate spam classification algorithms on realistic data.

To realize its full potential, the methodology requires a substantial investment of effort to implement and deploy a distributed, blind test infrastructure. For this thesis, time constraints forced us to build what amounts to a centralized simulation of the methodology. As a result, we were not able to conduct the kind of really large-scale study that we originally envisioned. Also, being centralized, we were not able to benefit from the efficiency of the distributed design. This hampered our efforts by limiting the number of experiments we could conduct each day. We believe that a full implementation of the methodology, deployed across a truly large variety of volunteers, would be well worth the effort (see Section 6.6).

Despite falling short of our ambitions, we believe that the results we did collect are the most relevant to date in terms of predicting the effectiveness of enterprise spam filters. No study we've seen has deliberately and systematically evaluated these algorithms on a variety of individual, complete streams of real email. This thesis accomplishes that, and shows, in particular, that both Naive Bayes and linear SVM classifiers are likely to perform very well in an enterprise environment. AdaBoost deserves another chance, using a more powerful implementation (see Section 6.6).

Specifically, a goal of building an enterprise filter that, on average, blocks greater than 90% of spam while suffering less than a 1% false positive rate seems feasible.

Also, in Appendix B, our statistical analysis shows that the variation in classifier performance between users is extremely significant; this validates our assertion that it is important to keep messages from different people separate when evaluating spam classifiers.

6.3 Managing Risk

The biggest risk for any spam classifier is the potential for loss of legitimate mail. While we have discussed practical ways to engineer protection against loss into our enterprise spam filter architecture (see Section 3.2), it is important to at least understand the magnitude of the risk, and better yet - to manage it.

Our results have shown that all three algorithms make it possible to tune filters, making trade-offs between effectiveness (spam blocking) and safety (minimization of false positives). Furthermore, these trade-offs appear to behave predictably; it appears possible to manage risk in a disciplined way. This means that an administrator could centrally set a desired false positive rate, which would cause particular decision thresholds to be used for each user (in general, it would be different for everyone, as determined by the training process on the client side). As long as everyone's filter key was kept up-to-date, the administrator could probably feel fairly confident that the overall average false positive rate for the enterprise as a whole would be fairly close to that desired setting. The administrator would also be able to predict fairly accurately the average spam recall for the enterprise as a whole (using training metrics reported back from each client). In summary, it appears possible that an enterprise spam filter with personalized filters could be centrally managed to achieve enterprise-wide spam blocking and risk targets.

Furthermore, our results show that the false positives that do occur tend to be among the least confident "spam" decisions. On average, for all three algorithms, roughly 90% of real spam scored higher on the "spamminess" scale than even the worst false positive. This makes it much easier to engineer loss-protection mechanisms. For example, there could be two decision thresholds instead of one; the second would be used to assign an "unsure" classification to a relatively small number of borderline messages, and these could be stored temporarily somewhere instead of blocked.

6.4 Bootstrapping

Machine-learning-based spam classification algorithms need a corpus of labeled messages to train on before they can be used. The question is: how large must that corpus be? That has direct practical implications. It affects how much effort and care users must put into sorting their mail during a start-up period. It also affects how long the period is between deploying an enterprise spam filter and benefiting from it.

Our results show that the Naive Bayes and linear SVM classifiers appear capable of excellent performance on corpora as small as 50 messages. For some people, that could mean a single day! In general, it appears that machine-learning-based enterprise spam filters can get up and running in a matter of days.

The use of some enterprise-wide "default" filter key could reduce the initial sorting effort required by users (analogous to what we did using Spam Assassin).

6.5 Summary

This thesis has provided strong evidence that a practical enterprise spam filter can be built using machine learning algorithms. Based on the architecture we have proposed, and our realistic test methodology, our results suggest that such filters would likely satisfy the critical requirements defined above.

The specific contributions of this thesis are:

- A proposed architecture for efficient and scalable enterprise spam filters that use machine-learning-based spam classification algorithms;
- A novel experimental methodology that overcomes the privacy barrier - making it possible to systematically evaluate spam classification algorithms on the email received by large numbers of users;
- The first evaluation, under that methodology, of three popular spam classification algorithms - yielding arguably the most useful performance results to date for those algorithms.
- Statistical evidence of the importance of personalization in the evaluation of spam classification algorithms;
- An empirical quantification of the inherent trade-off between the effectiveness and safety of those algorithms - showing how they enable enterprises to manage their risk in a disciplined way;
- A comparison of average algorithm performance versus corpus size - showing how many messages these algorithms typically need to see before they become useful;
- A comparison of average algorithm execution times - providing evidence that the proposed architecture is indeed efficient and scalable;
- An extensible C++ API for spam classification - supporting future studies based on the new methodology.

6.6 Future Work

There are many interesting directions for future research on machine-learning-based enterprise spam filters. First of all, a truly large-scale study involving tens or hundreds of volunteers would yield extremely valuable and more statistically significant data. For example, a study of this size would be needed to examine the effect of the proportion of spam a user receives on filter performance. Client plug-in software (for popular clients such as Microsoft Outlook) and a centralized server (to collect test reports from clients and provide them with updated experiments) would be needed. The spam classification API built for this study is designed for use in such a project.

Also, the AdaBoost algorithm deserves more attention. The implementation used in this study was somewhat crippled by using just decision tree stumps. Multi-level trees have been shown to provide better results [3], and would probably be fast enough (especially since we now know that execution times are dominated by message preprocessing tasks).

There are a number of fairly standard parsing techniques that are worth trying, such as decoding Base64-encoded text and stripping out HTML tags. These may well improve algorithm safety. It would also be worth trying out different tokenization strategies, including special treatment of email addresses, URLs and IP addresses.

In this thesis, we have stressed the importance of personalization. We have asserted that personalized filters are likely to be more effective and more safe than generic filters. It would be interesting to design an experiment that tests these assertions.

We have also stressed the importance of adaptability. It would be interesting to study how often these algorithms require re-training. This also opens a number of questions around the idea of what you re-train *on*. If the vast majority of spam is blocked, the proportion of spam in users' corpora will plummet. How do you choose messages to train on? Similarly, the role of active learning in general should be examined, especially in terms of minimizing user effort.

One characteristic our methodology shares with real life is noisy data. In real life, people will make mistakes when classifying their mail. An important question is how will effectiveness and safety vary with various levels of random noise inserted into the training set?

It may also be worth investigating the idea of training a second classifier on just the more-difficult "borderline" messages, where most of the false positives and false negatives occur.

Finally, in-depth efficiency and scalability analyses and simulations could be performed on a prototype of our architecture to strengthen the evidence for its feasibility.

Bibliography

[1] I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, G. Paliouras, and C. D. Spyropoulos. An evaluation of naive bayesian anti-spam filtering. In *Proceedings of the Workshop on Machine Learning in the New Information Age, 11th European Conference on Machine Learning (ECML 2000)*, pages 9–17, Barcelona, Spain, 2000.

[2] I. Androutsopoulos, G. Paliouras, V. Karkaletsis, G. Sakkis, C. D. Spyropoulos, and P. Stamatopoulos. Learning to filter spam E-mail: A comparison of a naive bayesian and a memory-based approach. In *Proceedings of the Workshop on Machine Learning and Textual Information Access, 4th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2000)*, pages 1–13, Lyon, France, 2000.

[3] Xavier Carreras and Lluís Márquez. Boosting trees for anti-spam email filtering. In *Proceedings of RANLP-01, 4th International Conference on Recent Advances in Natural Language Processing*, Tzigov Chark, BG, 2001.

[4] William W. Cohen. Learning rules that classify e-mail. In *AAAI Spring Symposium on Machine Learning in Information Access*, 1996.

[5] Nello Cristianini and Bernhard Schoelkopf. Support vector machines and kernel methods, the new generation of learning machines. *Artificial Intelligence Magazine*, 23(3):31–41, 2002.

[6] Yanlei Diao, Hongjun Lu, and Dekai Wu. A comparative study of classification-based personal e-mail filtering. In *Proceedings of PAKDD-00, 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 408–419, Kyoto, JP, 2000. Springer Verlag, Heidelberg, DE.

[7] Harris Drucker, Vladimir Vapnik, and Dongui Wu. Support vector machines for spam categorization. *IEEE Transactions on Neural Networks*, 10(5):1048–1054, 1999.

[8] Jose M. Gómez Hidalgo. Evaluating cost-sensitive unsolicited bulk email categorization. In *Proceedings of SAC-02, 17th ACM Symposium on Applied Computing*, pages 615–620, Madrid, ES, 2002.

[9] J. M. Gómez Hidalgo, M. Maña López, and E. Puertas Sanz. Combining text and heuristics for cost-sensitive spam filtering. In *Proceedings of the Fourth Compu-*

tational Natural Language Learning Workshop, CoNLL-2000, Lisbon, Portugal, 2000. Association for Computational Linguistics.

[10] Aleksander Kolcz and Joshua Alspector. SVM-based filtering of e-mail spam with content-specific misclassification costs. In *Proceedings of the TextDM'01 Workshop on Text Mining - held at the 2001 IEEE International Conference on Data Mining*, 2001.

[11] Kun-Lun Li, Kai Li, Hou-Kuan Huang, and Sheng-Feng Tian. Active learning with simplified svms for spam categorization. In *Proceedings of the International Conference on Machine Learning and Cybernetics (ICMLC02)*, pages 1198–1202, Beijing, China, 2002.

[12] R. Lyman Ott. *An Introduction to Statistical Methods and Data Analysis*. Duxbury Press, Belmont, California, 1993.

[13] Patrick Pantel and Dekang Lin. Spamcop: A spam classification & organization program. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin, 1998. AAAI Technical Report WS-98-05.

[14] John C. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines, 1998.

[15] Jason D. M. Rennie. ifile: An application of machine learning to mail filtering. In *Proceedings of the KDD-2000 Workshop on Text Mining*, 2000.

[16] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A bayesian approach to filtering junk E-mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin, 1998. AAAI Technical Report WS-98-05.

[17] Georgios Sakkis, Ion Androutsopoulos, Georgios Paliouras, Vangelis Karkaletsis, Constantine D. Spyropoulos, and Panagiotis Stamatopoulos. Stacking classifiers for anti-spam filtering of E-mail. In *Proceedings of EMNLP-01, 6th Conference on Empirical Methods in Natural Language Processing*, Pittsburgh, US, 2001. Association for Computational Linguistics, Morristown, US.

[18] Robert E. Schapire. A brief introduction to boosting. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1401–1406, 1999.

[19] Robert E. Schapire and Yoram Singer. Improved boosting algorithms using confidence-rated predictions. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, pages 80–91, 1998.

[20] Karl-Michael Schneider. A comparison of event models for naive bayes anti-spam e-mail filtering. In *Proceedings of the 10th Conference of the European Chapter of the Association for Computational Linguistics*, 2003.

[21] Nuanwan Soonthornphisaj, Kanokwan Chaikulseriwat, and Piyanan Tang-on. Anti-spam filtering: A centroid-based classification approach. In *Proceedings of the International Conference on Signal Processing (ICSP' 02)*, Beijing, China, 2002.

[22] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 2000.

Appendix A

Detailed Results

A.1 Feature Selection Experiments

Table A.1 shows, for the Naive Bayes classifier, the average spam recall measurements versus the number of attributes for several fixed false positive rates.

	Spam Recall(%)							
FPR(%)	50	100	250	500	1000	2500	5000	10000
5.0	88.9	94.2	97.5	98.4	98.9	99.0	99.2	99.2
2.0	79.4	89.6	95.1	97.4	98.3	98.5	98.8	98.7
1.0	75.5	86.1	93.1	96.4	97.5	97.7	97.9	98.2
0.5	70.6	83.1	91.7	95.4	96.3	96.5	96.7	97.6
0.0	66.0	77.9	89.8	92.0	93.9	94.2	94.0	93.9

Table A.1: Naive Bayes: spam recall vs. number of attributes.

Table A.2 shows, for the SVM classifier, the average spam recall measurements versus the number of attributes for several fixed false positive rates.

	Spam Recall(%)							
FPR(%)	50	100	250	500	1000	2500	5000	10000
5.0	96.4	98.9	99.1	99.2	99.4	99.4	99.3	99.4
2.0	86.6	96.0	96.8	97.8	98.8	98.9	99.0	99.0
1.0	74.6	93.5	93.7	95.9	96.8	97.2	97.1	97.3
0.5	61.9	88.6	90.0	92.6	93.4	94.1	94.8	94.6
0.0	40.6	69.2	81.3	88.0	91.3	90.1	90.1	90.8

Table A.2: SVM: spam recall vs. number of attributes.

Table A.3 shows, for the AdaBoost classifier, the average spam recall measurements versus the number of attributes for several fixed false positive rates (baseline configuration used except using 100 rounds).

	Spam Recall(%)							
FPR(%)	50	100	250	500	1000	2500	5000	10000
5.0	98.2	98.7	99.2	99.0	99.1	99.0	98.8	98.7
2.0	94.7	96.5	97.6	97.7	98.4	98.3	98.2	97.8
1.0	82.9	93.5	95.2	96.9	97.0	96.8	97.1	96.8
0.5	67.8	89.3	93.7	95.6	95.5	95.5	95.8	95.7
0.0	50.7	72.6	91.4	90.2	89.0	93.3	91.8	84.0

Table A.3: AdaBoost: spam recall vs. number of attributes.

A.2 AdaBoost Rounds Experiment

Table A.4 shows, for the AdaBoost classifier, the average spam recall measurements versus the number of rounds for several fixed false positive rates.

	Spam Recall(%)				
FPR(%)	25	50	100	200	400
5.0	98.6	98.8	99.2	99.1	99.1
2.0	96.1	97.0	97.5	97.4	97.7
1.0	93.5	94.9	94.6	93.9	95.0
0.5	89.1	92.4	91.8	91.8	93.1
0.0	67.3	85.0	84.3	86.1	84.3

Table A.4: AdaBoost: spam recall vs. number of rounds.

A.3 Feature Extraction Experiments

Tables A.5, A.6 and A.7 show, for the Naive Bayes, SVM and AdaBoost classifiers respectively, the average spam recall measurements versus the inclusion of header names for several fixed false positive rates.

	Spam Recall(%)	
FPR(%)	Baseline	With Header Names
5.0	99.0	99.0
2.0	98.3	98.4
1.0	97.5	97.8
0.5	96.1	96.9
0.0	93.7	94.2

Table A.5: Naive Bayes: spam recall vs. inclusion of header names.

	Spam Recall(%)	
FPR(%)	Baseline	With Header Names
5.0	99.4	99.4
2.0	98.6	98.5
1.0	96.6	97.1
0.5	94.8	95.4
0.0	91.7	93.0

Table A.6: SVM: spam recall vs. inclusion of header names.

	Spam Recall(%)	
FPR(%)	Baseline	With Header Names
5.0	99.0	98.7
2.0	97.6	97.2
1.0	94.2	95.0
0.5	92.1	89.7
0.0	83.6	75.6

Table A.7: AdaBoost: spam recall vs. inclusion of header names.

Tables A.8, A.9 and A.10 show, for the Naive Bayes, SVM and AdaBoost classifiers respectively, the average spam recall measurements versus the use of ”header tags” for several fixed false positive rates.

	Spam Recall(%)	
FPR(%)	Baseline	With Header Tags
5.0	98.9	99.1
2.0	98.4	98.0
1.0	97.7	96.6
0.5	96.7	96.0
0.0	94.0	92.9

Table A.8: Naive Bayes: spam recall vs. use of header tags.

Tables A.11, A.12 and A.13 show, for the Naive Bayes, SVM and AdaBoost classifiers respectively, the average spam recall measurements versus the inclusion of header values for several fixed false positive rates. Note that both configurations included the subject header.

Tables A.14, A.15 and A.16 show, for the Naive Bayes, SVM and AdaBoost classifiers respectively, the average spam recall measurements versus the inclusion of body text for several fixed false positive rates.

	Spam Recall(%)	
FPR(%)	Baseline	With Header Tags
5.0	99.3	99.3
2.0	97.9	98.3
1.0	97.3	95.9
0.5	96.5	94.2
0.0	93.2	88.5

Table A.9: SVM: spam recall vs. use of header tags.

	Spam Recall(%)	
FPR(%)	Baseline	With Header Tags
5.0	99.1	99.2
2.0	98.3	97.1
1.0	97.1	95.0
0.5	94.8	92.5
0.0	78.8	88.8

Table A.10: AdaBoost: spam recall vs. use of header tags.

	Spam Recall(%)	
FPR(%)	Baseline	Without Header Values
5.0	99.0	98.0
2.0	98.5	97.0
1.0	97.6	95.4
0.5	96.6	93.6
0.0	93.9	90.7

Table A.11: Naive Bayes: spam recall vs. inclusion of header values.

	Spam Recall(%)	
FPR(%)	Baseline	Without Header Values
5.0	99.4	98.5
2.0	98.1	97.4
1.0	97.5	92.2
0.5	94.4	87.1
0.0	91.9	78.6

Table A.12: SVM: spam recall vs. inclusion of header values.

	Spam Recall(%)	
FPR(%)	Baseline	Without Header Values
5.0	99.0	98.5
2.0	97.5	96.3
1.0	95.9	93.1
0.5	93.8	88.3
0.0	84.1	70.1

Table A.13: AdaBoost: spam recall vs. inclusion of header values.

	Spam Recall(%)	
FPR(%)	Baseline	Without Body Text
5.0	98.9	98.0
2.0	98.4	92.0
1.0	97.7	84.0
0.5	96.4	78.1
0.0	93.8	71.5

Table A.14: Naive Bayes: spam recall vs. inclusion of body text.

	Spam Recall(%)	
FPR(%)	Baseline	Without Body Text
5.0	99.5	98.2
2.0	98.3	94.4
1.0	97.7	88.7
0.5	95.1	81.9
0.0	92.0	67.4

Table A.15: SVM: spam recall vs. inclusion of body text.

	Spam Recall(%)	
FPR(%)	Baseline	Without Body Text
5.0	99.0	98.4
2.0	97.6	94.5
1.0	96.4	89.1
0.5	91.9	81.5
0.0	73.9	69.2

Table A.16: AdaBoost: spam recall vs. inclusion of body text.

Tables A.17, A.18 and A.19 show, for the Naive Bayes, SVM and AdaBoost classifiers respectively, the average spam recall measurements versus case-sensitivity for several fixed false positive rates.

	Spam Recall(%)	
FPR(%)	Baseline	Case-Insensitive
5.0	98.9	98.7
2.0	98.3	98.0
1.0	97.8	96.7
0.5	96.1	95.5
0.0	93.1	93.5

Table A.17: Naive Bayes: spam recall vs. case sensitivity.

	Spam Recall(%)	
FPR(%)	Baseline	Case-Insensitive
5.0	99.5	99.4
2.0	98.1	98.1
1.0	97.2	96.9
0.5	95.8	95.4
0.0	92.3	88.8

Table A.18: SVM: spam recall vs. case sensitivity.

	Spam Recall(%)	
FPR(%)	Baseline	Case-Insensitive
5.0	99.1	98.9
2.0	97.7	97.8
1.0	95.7	95.6
0.5	92.9	93.0
0.0	85.4	83.9

Table A.19: AdaBoost: spam recall vs. case sensitivity.

A.4 Corpus Size Experiments

Tables A.20, A.21, A.22, A.23 and A.24 show, for corpus sizes of 50, 100, 200, 400 and 800 messages respectively, the average spam recall measurements versus algorithm for several fixed false positive rates. Note that one of the seven users was omitted from these results because he did not have enough messages in his database to participate in the 800-message test. Also note that this explains the discrepancy between Tables 5.9 and A.23.

A sudden drop-off in performance can be seen in Table A.24 at the 0% false positive rate setting. This is the "worst case" setting; it requires the decision threshold to be set high enough that even the worst false positive does not get marked as spam. Therefore a single peculiar legitimate message (which might even be incorrectly labeled) can dramatically affect the overall results - especially since the average is across only six people. For this reason, the 0% false positive rate would likely be too unstable a setting for real-world operation.

	Spam Recall(%)		
FPR(%)	Naive Bayes	SVM	AdaBoost
5.0	95.7	97.6	59.6
2.0	92.8	94.2	44.4
1.0	91.9	92.9	38.5
0.5	91.4	92.2	35.6
0.0	90.9	91.6	32.6

Table A.20: Spam recall vs. algorithm for 50 messages.

	Spam Recall(%)		
FPR(%)	Naive Bayes	SVM	AdaBoost
5.0	97.2	98.4	81.0
2.0	95.6	95.6	73.0
1.0	94.8	93.7	63.9
0.5	94.4	92.7	59.2
0.0	94.0	91.8	54.5

Table A.21: Spam recall vs. algorithm for 100 messages.

	Spam Recall(%)		
FPR(%)	Naive Bayes	SVM	AdaBoost
5.0	98.3	99.1	95.8
2.0	95.9	94.9	89.0
1.0	93.3	93.1	83.4
0.5	92.3	92.1	75.3
0.0	91.4	91.1	67.0

Table A.22: Spam recall vs. algorithm for 200 messages.

	Spam Recall(%)		
FPR(%)	Naive Bayes	SVM	AdaBoost
5.0	98.7	99.5	98.5
2.0	97.7	97.5	97.0
1.0	96.8	96.0	93.5
0.5	95.3	94.9	90.5
0.0	93.3	92.7	87.7

Table A.23: Spam recall vs. algorithm for 400 messages.

	Spam Recall(%)		
FPR(%)	Naive Bayes	SVM	AdaBoost
5.0	98.7	99.3	98.9
2.0	97.4	98.4	97.7
1.0	95.9	96.9	95.1
0.5	93.1	94.5	92.3
0.0	75.0	75.9	84.2

Table A.24: Spam recall vs. algorithm for 800 messages.

Appendix B

Statistical Significance Analysis

The study only included seven users, so we did not expect to find statistically significant differences in effectiveness between the algorithms. In this chapter we perform significance testing on the "main" experiment - the one that compares the best configuration of each algorithm on a corpus size of 400 messages.

Table B.1 shows the spam recall measurements for each of the seven users, at a 1% false positive rate, on 400 messages. The users are given the labels 'A' to 'G'.

	Spam Recall(%)		
User	Naive Bayes	SVM	AdaBoost
A	99.7	100.0	100.0
B	99.6	100.0	100.0
C	95.7	96.9	91.3
D	98.2	96.1	92.7
E	96.1	97.7	96.0
F	96.8	99.4	95.6
G	94.5	85.8	85.6

Table B.1: Spam recall at 1% false positive rate for each user.

The differences between the mean spam recall values for the three algorithms can be analyzed using two-factor Analysis of Variance (ANOVA), without replication [12]. In our experiment the factor of interest is the algorithm. We have added a second factor - the user - as a "blocking" factor. This is because the between-user variances could obscure (or "block") the between-algorithm variances in a one-factor ANOVA. The two-factor ANOVA is more powerful - it can distinguish between the two sources of variance.

Using Microsoft Excel's "ANOVA: Two-Factor Without Replication" function with an alpha of 0.05, the data in Table B.1 produces the output shown in Tables B.2 and B.3.

ANOVA: Two-Factor Without Replication

SUMMARY	Count	Sum	Average	Variance
A	3	299.66	99.89	0.039
B	3	299.60	99.87	0.053
C	3	283.90	94.63	8.815
D	3	286.98	95.66	7.874
E	3	289.79	96.60	0.931
F	3	291.74	97.25	3.701
G	3	265.81	88.60	25.820
Naive Bayes	7	680.52	97.22	3.982
SVM	7	675.82	96.55	25.010
AdaBoost	7	661.14	94.45	26.199

Table B.2: Two-factor ANOVA analysis (part 1)

ANOVA

Source of Variation	SS	df	MS	F	P-value	F crit
Rows	265.88	6	44.31	8.15	0.0011	3.00
Columns	29.20	2	14.60	2.68	0.1087	3.89
Error	65.26	12	5.44			
Total	360.34	20				

Table B.3: Two-factor ANOVA analysis (part 2)

The number of interest is the p-value of about 0.11 for the between-algorithms variation; since this is much larger than our alpha of 0.05, we cannot reject the null hypothesis that the differences between the algorithm means are not significant at the 0.05 level. As expected, the number of users is simply too small. Note also that the ANOVA assumption of equal variances would have to be checked before assuming significance - and Table B.2 shows variances that appear quite different.

Similar results are obtained for the mean spam recall at 0%, 0.5%, 2% and 5% false positive rates.

There is however, another interesting observation to make: the between-user variation is extremely significant. This is solid evidence to back up our claim that mail from different people should not be mixed together for training. Classifier performance is not just affected by the choice of algorithm; clearly it is also strongly affected by who the messages come from. This between-user variation could obscure the between-algorithm variation if mail was not kept separate.

Appendix C

Glossary

Attribute Used interchangeably with “feature”. See Section 2.2.

Classified Used interchangeably with “labeled”. See Sections 2.1 and 4.2.

Classifier An implementation of a machine-learning-based spam classification algorithm. See Section 2.1.

Corpus A set of labeled documents that is used for training and/or testing a classifier. It may be subdivided into a training set and a test set, or it may be divided up into several folds for cross validation. See Sections 2.1, 2.8 and 4.3.3.

Cross validation See Section 2.8.

Database Physical files storing labeled copies of all messages received by a SpamTest volunteer tester. A subset of the messages may be chosen as the corpus for a particular experiment. See Sections 4.2 and 4.3.3.

Dictionary See Section 2.4.

Document Used interchangeably with “message”. Refers to an RFC 822 compliant email message. See Section 2.2.

Enterprise spam filter See Chapter 1.

False positive rate The proportion of legitimate messages misclassified as spam.

Feature Used interchangeably with “attribute”. See Section 2.2.

Feature selection See Section 2.4.

Feature extraction See Section 2.3.

Filter A mechanism for blocking or flagging spam. May be implemented by a classifier.

Filter key See Section 3.2.

Generalize See Section 2.1.

Instance mapping See Section 2.5.

Instance vector A vector representation of a "document". See Section 2.2.

Labeled This means that the correct class (that is, spam or not-spam) is stored with the message. See Sections 2.1 and 4.2.

Message Used interchangeably with "document". Refers to an RFC 822 compliant email message. See Section 2.2.

Overfitting See Section 2.1.

Term Used interchangeably with "word" or "token". See Section 2.2.

Test Set A set of labeled documents used to test a trained classifier. See Section 2.1.

Threshold See Section 2.6.

Token Used interchangeably with "word" or "term". See Section 2.2.

Training Set A set of labelled documents used to train a classifier. See Section 2.1.

Spam precision The proportion of all messages classified as spam that actually are.

Spam recall The proportion of spam messages that are correctly classified.

Stratification See Section 2.8.

Word Used interchangeably with "term" or "token". See Section 2.2.