Institutionen för datavetenskap
Department of Computer and Information Science

Examensarbete

Spam filter for SMS-traffic

av

Johan Fredborg

LIU-IDA/LITH-EX-A—13/021-SE

2013-05-16

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköpings universitet
Institutionen för datavetenskap

Examensarbete

Spam filter for SMS-traffic

av

Johan Fredborg

LIU-IDA/LITH-EX-A—13/021-SE

2013-05-16

Handledare: Olov Andersson, Fredrik Söder
Examinator: Fredrik Heintz

Abstract

Communication through text messaging, SMS (Short Message Service), is nowadays a huge industry with billions of active users. Because of the huge user base it has attracted many companies trying to market themselves through unsolicited messages in this medium in the same way as was previously done through email. This is such a common phenomenon that SMS spam has now become a plague in many countries.

This report evaluates several established machine learning algorithms to see how well they can be applied to the problem of filtering unsolicited SMS messages. Each filter is mainly evaluated by analyzing the accuracy of the filters on stored message data. The report also discusses and compares requirements for hardware versus performance measured by how many messages that can be evaluated in a fixed amount of time.

The results from the evaluation shows that a decision tree filter is the best choice of the filters evaluated. It has the highest accuracy as well as a high enough process rate of messages to be applicable. The decision tree filter which was found to be the most suitable for the task in this environment has been implemented. The accuracy in this new implementation is shown to be as high as the implementation used for the evaluation of this filter.

Though the decision tree filter is shown to be the best choice of the filters evaluated it turned out the accuracy is not high enough to meet the specified requirements. It however shows promising results for further testing in this area by using improved methods on the best performing algorithms.

This work could not have been done if not for the opportunity given to me by the people at the company Fortytwo Telecom. They gave me the prerequisites needed to test these technologies and gave me a good idea of the requirements for a filter like this.

I want to thank Fredrik Söder at Fortytwo Telecom for being my contact person to bounce ideas back and forth to. I also specifically want to thank my supervisor Olov Andersson for having been a great help in giving me answers to all the questions I could not answer myself, for pointing me in the right direction of where to take this work and for being a big help in the writing process of this thesis.

And lastly I would like to thank my examiner Fredrik Heintz for making this possible and taking on responsibility for the work!

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This work is a study together with Fortytwo Telecom (www.fortytwotele.com) into the applicability of established classifiers acting as filters for unsolicited messages (e.g. spam) in SMS (Short Message Service) communication. As the reliance on communication by email has steadily increased in society, spam mails have become a bigger and bigger problem and the necessity to have accurate and fast spam filters is almost considered a must now for any email provider to provide a good service to its customers. As the market for text-messaging between mobile phones has grown, the spam advertising phenomena has also spread over to this market.

The algorithms tested to be used as filters are in the machine learning algorithms branch of artificial intelligence. These algorithms are built to learn from data and then apply what it has been taught on previously unseen data. This is a technology which has had much practical use for many years now for spam filters and is still heavily researched. Some of these algorithms nowadays have been applied to email services on the Internet to protect their users from becoming overflowed by unsolicited messages and this study will see if they can also be successfully applied to the SMS domain.

This domain has different characteristics and requirements from filters used by email providers. SMS-messages always contains very little data to analyze in comparison to emails. They are also normally paid for by the sender, they are expected to always arrive after being sent and they are expected to arrive relatively quickly. Emails on the other hand are normally not paid for, the urgency is not as critical and it is not completely uncommon that legitimate emails are caught in spam filters. The purpose is therefore to evaluate the applicability of established machine learning algorithms tested for filtering spam in email messages, and instead test them on this domain.

1.1 Background

Fortytwo Telecom is a "*leading telecom service provider and an SMS gateway provider offering good-quality mobile messaging services worldwide.*" [8]. They were interested in examining the accuracy and speed of different types of spam filters than their current ones and showed an interest in statistical classification and interpretable classifiers like decision trees.

After meeting and discussing what they wanted to achieve, we eventually agreed on four classifiers that were going to be evaluated. The classifiers were a naïve Bayes classifier, a decision tree classifier a support vector machine and a dynamic Markov coding classifier. They had a few concrete requirements for how they should perform, namely that the filters could not use more than 1 GB of memory, at least 10.000 messages should be possible to filter per second, and lastly that not more than around 0.1% of all non-spam messages were allowed to be stopped by the filter.

1.2 The Mobile Phone Network

The network which the data travels through for SMS communication consists of many parts. To get an overview of where a solution for an SMS spam filter might be applied the major parts of a GSM network is explained briefly. The most relevant parts and terminology to this work would be the BTS (Base Transceiver Station), the BSC (Base Station Controller), the MSC (Mobile Switching Centre), the SMSC (Short Message Service Center), the VLR (Visited Location Register), the HLR (Home Location Register) and the EIR (Equipment Identity Register).

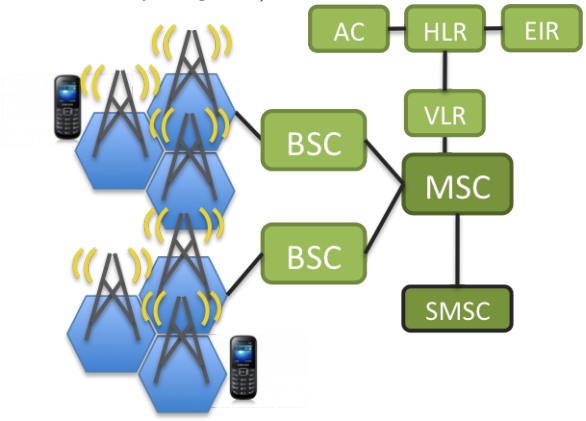

Figure 1.1: An overview of a GSM network.

1.2.1 Communication Infrastructure

The antennas in figure 1.1 is the base transceiver station which normally consists of an antenna and radio equipment to communicate with mobile devices and its base station controller. The base station controller is commonly responsible for many transceiver stations at the same time, and amongst other things forwards communication coming from mobile devices through a transceiver station to the mobile switching centre as well as communication coming from the mobile switching centre to a mobile device. It also handles the handover between different base transceiver stations if a mobile device moves between different cells of the network.

Just as the base station controller is responsible for handover between different transceiver stations, the mobile switching centre is responsible for handovers between different base station controllers if a mobile device enters a cell with a transceiver station not under the current base station controller's control. Besides handovers it is is also responsible for making connections between a mobile device to other mobile devices or the PSTN (Public Switched Telephone network), the normal phone network for stationary phones. The base station controller is connected to several databases such as the visited location register, home location register and equipment identity register.

1.2.2 Databases

The visited location register keep information about the current whereabouts of a specific subscriber inside the area it is responsible for to be able to route a call to a mobile device through the correct base station controller. The home location register keeps information about a subscriber's phone number, identity and the general location amongst other things. The equipment identity register keeps information about all mobile devices IMEI (International Mobile Equipment Identity) which is unique for each mobile device - that *should* be either banned or tracked if for example the device would be stolen. The most relevant part for this study is the short message service center. This service is responsible for storing and forwarding messages to a recipient. This is done on a retry-schedule until the message is finally sent to the recipient. If it is sent successfully the recipient returns an acknowledgement and the message is removed from the system. If enough time has passed and the message has become expired without a successful delivery, the message is also removed. This is the service where a SMS filter could likely reside.

1.3 Purpose

The purpose of this Master's thesis is to evaluate several established machine learning algorithms in a mobile phone text-messaging domain where the

length of a SMS (Short Message Service) in this domain is limited to the size of 140 bytes. The maximum number of characters would normally vary between 70 to 160 depending on the encoding chosen by the sender. The best algorithm according to the evaluation should be implemented and tested on real data from Fortytwo Telecom. The main challenges are to find an algorithm which filters messages fast enough, has reasonable space requirements and has a considerably low false positive compared to its true positive.

1.4 Limitations

No more than four algorithms are going to be evaluated. The algorithms were chosen during the pre-study by mainly comparing and trying to find the algorithms with the best accuracy as well as discussions with the supervisor. Though the naïve Bayes algorithm was chosen partly because of its simplicity and its tendency to be used as a baseline to other algorithms in studies. The DMC algorithm was chosen not only because of its high accuracy shown in some of the literature study [2] but also because of how it stands out from the other algorithms in how the messages are processed. This is explained at more detail in the chapter Learning Algorithms.

It was decided to only do deeper tests on one configuration of each type of algorithm evaluated because of time constraints. This means that each algorithm is tested with a varying number of features used such as 500, 1000, 1500 or 2500. Each test also vary the size of n-grams used. Either unigrams are used in the test, or unigrams combined with bigrams. Some of the algorithms can have their own specific configurations as well, these specific settings did not change but instead used a common configuration for each test. In total there were four different algorithms evaluated and each were trained and tested with three different sizes of available tokens for the algorithm.

The experiments were not done on the targeted server hardware. Therefor the requirement that at least 10.000 messages had to be filtered per second could not be strictly evaluated but performance of the classifiers is taken into account in the analysis. A lower amount could be acceptable after discussion. The framework which ran the experiments was also not optimized for speed. A careful implementation might increase the computational performance.

1.5 Prestudy

Before beginning this work it was necessary to get an overview of other studies evaluating machine-learning algorithms in this field. It was found that using machine learning algorithms is a very popular method to try to

specifically combat the problem of spam for emails and that many of these filters could give an accuracy in classification of above 90% [1].

It was found that emails have a slightly different message structure in comparison to SMS messages such as containing a subject field as well as a message body which may contain HTML markup, graphical elements and pure text. SMS messages on the other hand simply have a message body typically with pure text. These differences would not change the possibility to evaluate spam filtering for SMS any different than for emails, though it shows that emails may have more data to analyze in a single message. This was assumed to be not only for bad, since less data to analyze should increase the processing speed. But it also likely decreases the accuracy by having less information to base a decision on. Also as mentioned in section 1.3 an SMS message has a very limited amount of data that it can contain in comparison to emails.

During the study we found that there had been many evaluations performed for different machine learning algorithms in the email message domain but the single focus for most of these had been accuracy without unfortunately comparing the processing speeds.

There were several potential filters that showed a high accuracy, commercial as well as non-commercial such as the Bogofilter. After comparing evaluations from earlier works for filtering emails the filters of interest was narrowed down to a few non-commercial ones. Support vector machine, dynamic Markov coding(DMC) and prediction by partial matching(PPM) all showed some of the best results in several evaluations [2] [4]. Of these three DMC and PPM both were very similar in approach, however DMC most often showed a higher accuracy thus it was decided to not use PPM in favor of DMC.

The third algorithm chosen was the C4.5 algorithm [18]. The C4.5 algorithm was said to give average results on text classification problems [14]. What was intriguing about this algorithm however was its clear presentation in its output. The algorithm outputs a decision tree which is then used when filtering messages. This presentation makes it very easy for experts and non-experts alike to quickly understand what the filter is doing at any time. This simplicity of the filter could be helpful if non-experts are supposed to maintain it.

The last algorithm chosen for evaluation was a naïve Bayes algorithm. This is because it was found to commonly be used as a baseline for accuracy during these types of evaluations.

1.6 Methodology and Sources

The first step of this thesis was a literature study to find out which filters showed the best performance in the email-domain of messages. The literature study also aimed to find a good experimental methodology for how to compare spam filters.

If free existing implementations of these algorithms were found, these would be utilized to test the algorithms that were candidates for implementation. Otherwise an implementation of the algorithm will have to be done.

To compare the performance of the different algorithms, they were evaluated on four different metrics. How many messages per minute that could be filtered through each algorithm to see if the algorithm would be fast enough as a filter. How much memory that was consumed for a loaded filter with no workload, this was necessary so the memory limit was not exceeded. It was also interesting to know how fast a new filter could be trained from a set of training data if a filter needed to be updated. Lastly the most important metric was the accuracy so that the filter would not misclassify too many messages.

Each result from batches of messages filtered, were plotted with a ROC curve to analyze how well each filter classified or misclassified the batches of messages and to find a possibly optimal classification threshold for each model. ROC curves are discussed in section 1.6.4.

For a better statistical accuracy on the tests of the filters and to compute the confidence intervals for the ROC curve, a k-fold cross-validation discussed in section 1.6.3 was used for each of configuration. By using k-fold cross-validation it also means that less data of classified messages was needed for achieving a strong statistical accuracy and thus less time classifying the data.

1.6.1 Methods of Measurement

Several methods are typically used for comparing results of classifiers. Some relevant methods here are precision, recall, accuracy, true positive, false negative, true negative and false negative-rates.

True positives from a classifier is spam classified correctly as such, and false positives would be non-spam classified incorrectly as spam. Conversely false negatives are spam not classified as such and true negatives would be a message that is correctly identified as such.

Precision, recall and accuracy, where tp stands for true positive, fp for false positive; tn for true negative and fn for false negative are defined as:

- $Precision = \frac{tp}{tp+fp}$
- $Recall = \frac{tp}{tp+fn}$
- $Accuracy = \frac{tp+tn}{tp+tn+fp+fn}$

In this study, precision tells the fraction of the message classified as spam to actually be spam. Precision needs to be very high if many messages are blocked, otherwise too many legitimate messages would be blocked as well. The other possibility is that very few messages are blocked, and in that case

the precision could go down as well and still not block too many legitimate messages. Of course the former one is sought for.

Recall is here the fraction of the spam messages that are actually classified as such, which of course should preferably be as high as possible to stop as many of them as it can.

Accuracy shows the fraction of messages that are correctly classified. It is important that the accuracy is high to have a low amount of legitimate messages be wrongly classified as well as to catch as much spam as possible.

These Methods as well as processing time gives us the necessary information needed to properly compare the different filters in respect to hardware requirements as well as the classification rate.

1.6.2 WEKA

WEKA (Waikato Environment for Knowledge Analysis) [11] is a machine learning framework and a possible candidate to test several of the algorithms on. WEKA is a program written in Java, which contains tools for a testing environment is supplied with many different machine learning algorithms. This free framework has very customizable tools for reading and pre-processing data for the machine learning algorithms. It also allows representing results from evaluations of algorithms through graphs, pure text as well as other means such as graphical presentations of the result of some of these algorithms such as decision tree constructs.

WEKA was used to read stored SMS communication data from a file and create a structured presentation from it which the learning algorithms could understand. It was also used to evaluate how well the performance was by measuring their time to completion as well as accuracy of each filter. These results were then stored by saving several time stamps between start and finish as well as saving special graphs called ROC curves showing the accuracy of each evaluated filters. ROC curves are mentioned more thoroughly in section 1.6.4.

1.6.3 K-fold Cross-validation

K-fold cross-validation splits data into k subsamples(or folds) where k is the amount of folds wished for. While one subsample is used for evaluating the filter, the other k-1 subsamples are used for training, this way the training data can still remain relatively large. This is done k times so that each subsample will be used for evaluation once and in the rest of the folds it will be used for training the filter.

The results from the training data can be averaged to get an estimation of the filters performance, both as ROC curves but also to average the speed of classification or training. Ten folds were used the experiments in this study when evaluating the filters.

1.6.4 ROC Curve

A ROC curve (Receiver operating characteristic) is a plot typically used to show the performance of how accurate a classifier's classification rate is. The axes correspond to the true positive rates(*tpr*) and false positive rates(*fpr*) respectively. The x-axis is based on the *fpr* which could represent for example legitimate messages classified as spam. The y-axis for *tpr* representing for example spam classified correctly as spam. The plot is built by having classified test data cases scored and successively decreasing a classification threshold value to compute new points in the plot based on current *fpr* and *tpr* values. A high score shows it is very likely that the training data in this case could be spam while a low score tells us it is most likely legitimate. The threshold value decides if a case will be marked as spam or as a legitimate message depending on if the classification score is higher or equal to the threshold, or if its not.

To properly use ROC curves for comparing the accuracy of different classifiers, the variance must be taken into account. If we are using k-fold cross-validation we get k results of test performance, one for each fold. Because of these several test cases we can extract a variation when generating the finalized ROC curve.

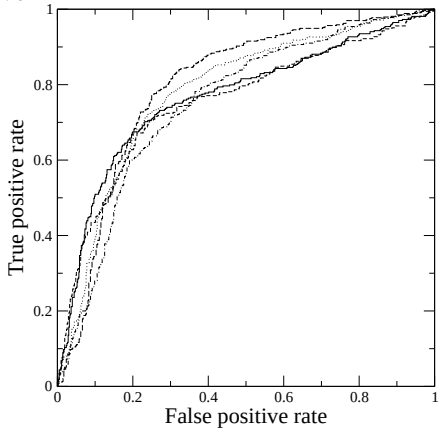

Figure 1.2: Multiple curves plotted by data from independent folds. Figure is from [6].

But simply merging the resulting curves created by the test cases from a cross-validation to plot the finalized ROC curve removes the possibility to take the variance into account, which is one of the reasons to even have several test folds.

To do this we need a "*method that samples individual curves at different points and averages the samples*" [6].

One method to do this is called *Vertical Averaging* which is appropriate to use when we can fix the *fpr*. In this case we can control it to an extent. It has fixed *fpr* values and as the name implies averages the *tpr* of each

curve from the test fold for each given *fpr* value. The highest *tpr* values are extracted from each fold for each *fpr* value and if a corresponding *fpr* value does not exist in the curve, the *tpr* value will be interpolated between the existing points. The *tpr* values for the given *fpr* are averaged and stored. And so the *fpr* value is incremented by a set amount and the same procedure is repeated again. So essentially for each given *fpr* value in the plot, the *tpr* value is given by the function $R(fprate) = mean[R_i(fprate)]$ where R_i is each ROC curve generated from the test folds.

By having the averaged ROC curve and the generated ROC curves from each test case, we can now find the variance for the *tpr* for each given *fpr* and the result could be something like in figure 1.3.

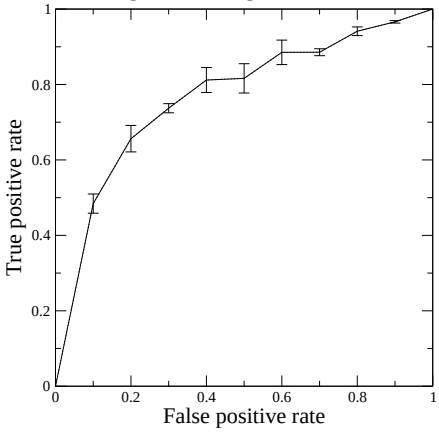

Figure 1.3: ROC curve plotted with confidence intervals computed by vertical averaging based on data from multiple folds. Figure is from [6].

In a ROC curve, one of the measurements used to compare the general performance would be by comparing the AUC (Area Under Curve), and the larger the area is the better. But many times such as in this study it may be interesting to only study a specific area of the curve. In this study it is important for the *fpr* to be in the range around zero to a half percent to be near the rate of 0.1% acceptable for legitimate messages of being filtered. So therefor the area of interest is at the very beginning of the curves.

1.7 Thesis Outline

Altogether there are six chapters and two appendices in this report. The first chapter gives a short summary about what the thesis intends to accomplish.

Chapter two intend to explain common preprocessing of messages which is shared by all but one of the filters.

Chapter three gives an overview of each of the classifiers used in this study.

Chapter four examines the results from the experiments done by testing each of these classifiers as SMS spam filters and concludes which classifier was the most suitable among them.

Chapter five explains the implementation of the filter which showed the best performance in the experiments and validates its performance.

The last chapter gives a discussion about the study and talks about future improvements.

Chapter 2

Preprocessing

Most spam filters for email services today incorporate several different layers of filters. The simplest level of filtering is the whitelisting and blacklisting of email addresses or specific words in a message - set by the operator. These are used to either keep or filter incoming email-messages.

Another layer may for example use specific predefined sentences or email addresses set by the user to once again decide if a message should be kept or not. If a message reached this layer the message must have already passed every layer above it first, and to reach the user it would normally need to go pass every level.

The layer this thesis is going to focus on is a layer where the whole message is more thoroughly scanned for patterns of spam in the text and by training the filter on already classified spam data to find these specific patterns. The most common algorithms used for this type of filter are from the field of machine learning algorithms, which can be seen in the amount of successful commercial applications applying them and these are the type of filters that will be evaluated in this thesis.

2.1 Overview

Of the four evaluated algorithms used for spam filtering, naïve Bayes, a C4.5 implementation of decision tree learning and support vector machines all require pre-processing both when training the filter and when classifying the incoming messages. Because of the nature of these three algorithms, they need each message to be represented structurally somehow to train and classify them. The bag-of-words, also referred to as the vector-space model was chosen for the representation and it is one of the most common approaches for this type of problem [10].

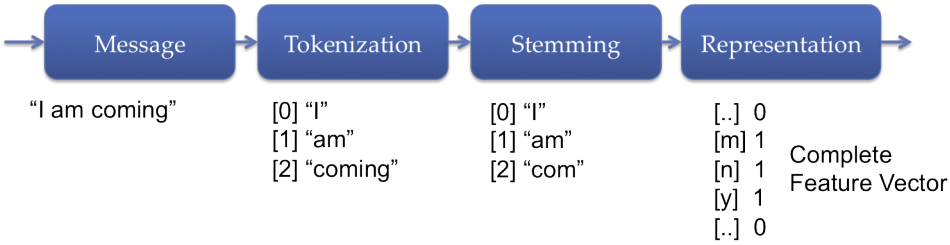

Figure 2.1: An overview of the pre-processing stage.

There are about four major steps used for pre-processing a message and create a proper representation of it that the classifiers can understand. As seen in the figure, it is the incoming message, the tokenization where messages are split up into words, the stemming where only the roots of words are kept and lastly the representation where each word is fitted into a possible slot in a feature vector.

2.1.1 Message

There are a few but major things that can structurally differ between messages apart from the content. There is the possible difference in encoding of a message for the computer. The choice of encoding can both decide the character space, but also the size and complexity for representing different characters.

There are many character encodings, so it is important to agree on a common encoding when communication to be able to properly read the contents of an incoming message. Some of the most common encoding are ASCII and UTF-8, while in this work the UCS-2 encoding were to be used for all incoming SMS-messages. It is a simple fixed-length format which uses 2 bytes to represent each character. Except for latin symbols it supports most other modern languages today to a varying degree from Arabic script to Chinese and Korean.

The other major part except the encoding of a message is the language it is written in. Depending on the language there may be many parts in a filter that could fail. If for example the filter is trained to analyze the contents of Swedish text messages but the message arriving is written in English, the filter might be confused without former training data of this.

2.1.2 Tokenization

Tokenization is the step in the processing where a complete message is divided into smaller parts by finding single characters or longer patterns which corresponds to a set delimiter. As is seen in figure 2.1 the message"*I am coming*" has been divided into the tokens *I*, *am*, *coming* by the whitespace delimiter.

For the tokenization of messages it was to be assumed that the messages being filtered were from roman text. This was an important assumption since some languages are very difficult to tokenize because of their structure. The Chinese written language for example do not necessarily use as many common obvious delimiters as is used in the latin alphabet(i.e. whitespace or an interrogation point). Therefor it is difficult to tokenize a sentence, because the sentence could just be one single long word.

The delimiters being used were adapted from this assumption. To find relevant tokens from this text, common delimiters for words such as space and commas were used, these are shown below.

\r \n \t . , ; : ’ ” ( ) ? !

N-grams

N-grams are a combination of N items from a sequence, in this case a sequence of written text of words or characters. The idea is to create a bigger word space to be able to attain more information by finding words that are commonly next to each other in a specific type of message. The size of N-grams in filters are usually between one(unigram), two(bigram) or three(trigram). While nothing is stopping us from having larger n-grams than that, the word space seems to become unmanageable and you need more training data.

In the evaluation of the performance of different filters in this thesis, word n-grams up to the size of two were used for the three filters using feature vectors to function, to try and find an optimal configuration for these.

2.1.3 Stemming and Letter Case

Stemming is an optional step after the tokenization of a message where each word token of the message is reduced to its morphological root form [4]. A problem with stemmers though is that they are dependent on the language they are created for which could give wrong or no results if a message of another language was incoming. An example is shown below how different words are all stemmed to the same morphological root form, which is the word ’catch’. **Catch**ing == Catch, **Catch**ed == Catch, **Catch**er == Catch. Choosing if upper- and lowercase representation of the same letter is distinct or not is another of the optional steps in the pre-processing with the same goal as stemming. This goal is to reduce the word space dimension as well as to improve the prediction accuracy for classifiers by overcoming the data sparseness problem in case the training data may be too small in comparison to the word space dimension [4].

Stemming of words and upper- and lowercase representation of letters has shown ”*indeterminate results in information retrieval and filtering*” [4] for spam filtering of email. I decided to use only lower case letters in the token representation to shrink the word space, while stemming of English

words were used in half of the tests to analyze which token representation showed best performance.

2.1.4 Weaknesses

A Clear weakness when using feature vectors is the possibility for a malicious sender to avoid words getting caught in the feature vector by spelling common spam words such as "Viagra" in different fashions. For example the character 'I' can be represented in at least 12 different ways "I, i, 1, l, ‖, ï, ì, í, :, *Ì*, *Í* or ĩ". The whole word can also be altered by inserting extraneous characters such as V_i_a_g_r_a, which makes the total amount of combinations at least 1,300,925,111,156,286,160,896 [12].

Another method used to try and decrease the effectiveness of a filter depending on if it is a bayesian filter would be Bayesian poisoning. This method aims to input specific words into the sent message, which would degrade the accuracy of the of the classifier. The person, likely a spammer, trying to cause a Bayesian poisoning would in that case try to fill the message with words commonly not found in a spam message. This would be to mask the real message among these other, for the spammer, more favorable words.

2.1.5 Representation

The representation is how a message should be formatted so that the classifier can understand and analyze what distinctive properties each message has. A typical example of this it the bag-of-words representation which is used in this work.

The representation is a N-dimensional feature vector, that is a vector with each axis(i.e. feature) representing a specific word or longer sentence and the value for each feature depending on if there is token in the message corresponding to it or not. There are two common representations usually used for the feature vector, firstly the binary approach were each feature of the feature vector either just represents if a word or sentence exist in the message or not (0,1), and secondly the numeral feature vector which shows a count of how many times the word or sentence appeared in said message (0, 1, 2, ...).

As can be seen in figure 2.2, the binary feature vector either has the count 0 or 1 for each feature, while the numeral feature vector keeps a count of the total hits for each feature. We can see that the message has 2 tokens corresponding to the feature *three* and the numeral example represents this as expected. All the other features that did not correspond to any token in the message is set to its default value of 0.

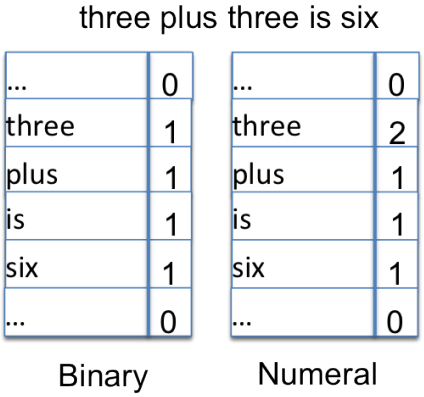

Figure 2.2: Showing the simple difference of the result for the message "three plus three is six" when using a binary and a numeral representation.

Once all the training data is pre-processed by the filter each feature will be assigned a permanent position in the feature vector so that any classifier relying on the representation always get an identical representation for each incoming message - apart from difference in feature counts of course which will by the only thing varying for each feature vector.

A spam filter might train on a huge amount of messages and the feature vector may grow larger in dimension with each message to be able to represent every single token from each and every message. This will be a problem not only for saving space and memory, but also for a definite loss of performance in speed in many filters when the vector grows too large as well as the need for more training data with more features. To combat this and reduce the dimension of the feature vector, a feature selection is used [4].

Feature Selection

A feature selection is used to decrease the number of total features in the representation. The goal of the feature selection is to both decrease the number of features in the representation as well as to keep the most relevant ones needed for a good classification result. These more relevant features usually distinguish themselves by appearing much more often in one class of messages than another. For example a word such as *PRIZE* appearing likely much more often in spam than in normal messages. Knowing this it could then possibly be chosen as on of the features that should be kept.

It was decided to use information gain as the feature selection algorithm in this thesis, as it is both commonly used and simple to understand. Information gain give out a rating for how well a certain feature in the feature vector can sort messages. For example into a collection of spam messages and a collection of legitimate messages by comparing the entropy in the original collection with the entropy in the following collections following a

split on the feature. The less mixed the collections are between legitimate and spam messages, the higher the rating will be.

Entropy which is used to find the information gain for a feature, can be seen for this work as how homogeneous a collection of classified messages is. The definition of entropy is "*A measure of "uncertainty" or "randomness" of a random phenomenon*" [13]. A low entropy means that a collection is very homogenous, while a high entropy means that it is more heterogeneous.

For the feature selection we want to find those features that splits our collection of messages into as homogeneous collections as possible. If we have a collection of for example both spam and legitimate messages, we want to find a feature that is common in spam messages and is not common in legitimate ones. As an example we can assume that *PRIZE* is common in spam, and we want to take all the messages with the word *PRIZE* in them to one collection, and the rest of the messages to another.

Type	PRIZE
Spam	Yes
Legitimate	No
Legitimate	Yes
Legitimate	No
Spam	Yes
Spam	Yes
Spam	Yes
Spam	Yes

Figure 2.3: A collection of classified messages of either type Spam or Legitimate and which either contains the feature PRIZE or not.

Now we get two collections, and if our assumption was correct, one collection should have a higher rate of spam and the other collection should have a higher rate of legitimate messages than in the original collection. To calculate the entropy, the formula $H(X) = -\sum_{i=1} nP(x_i) * log_b(P(x_i))$ is used where n in our case is the number of outcomes(which is for us *spam* or *legitimate*). $P(x_i)$ here is the probability for a feature x to belong to class i and the log base b is chosen to be 2. To get the original collections entropy we calculate

$$H(Spam) = -\frac{5}{8} * log_2(\frac{5}{8}) - \frac{3}{8} * log_2(\frac{3}{8}) = 0.95$$

and the new entropy if we split the collection on the word PRIZE we get

$$H(PRIZE = Yes) = -\frac{5}{6} * log_2(\frac{5}{6}) - \frac{1}{6} * log_2(\frac{1}{6}) = 0.65$$

and

$$H(PRIZE = No) = -\frac{0}{2} * log_2(\frac{0}{2}) - \frac{2}{2} * log_2(\frac{2}{2}) = 0$$

From the results we can see that the two new collections have a lower entropy than the original collection, which indicates that the feature *PRIZE* might be a good feature to select. This is a simplified example where only binary values for the features are used and thus only one test necessary per feature. For a numerical feature it would be necessary to split on each available value of the feature like $x <=$ and $x >$ where x is a value that the feature might have.

The formula for information gain is $IG(T, \alpha) = H(T) - H(T|\alpha)$. This is the expected reduction in entropy of the target distribution T when feature a is selected. $H(T)$ is the entropy for the original collection, and $H(T|\alpha)$ is the entropy of the new collections following the split on α. So as long as the information gain is larger than zero if means that if a split would occur on that certain feature, the new collections would be more homogenous than the original collection.

The features were all of either binary or continuous values, where for binary values only one run-through for the information gain to be had is necessary, while for the continuous features, tests will need to be done on all possible binary splits of the feature. For example if the range of values it can take is 1, 2 and 3, two tests on the same feature are necessary to check the information gain of binary split. One would be for {1} and {2,3} and one would be for {1,2} and {3}.

After having calculated the information gain for each available feature, it is easy to rank the features from highest score to the lowest to find those which best represents either a spam or legitimate message. When this is done a limited amount of features will be chosen which hopefully best represent each of the different types of messages.

2.2 Summary

The data for text representation can be encoded in many different ways and for different purposes, depending on what alphabets you wish to support. In this work a 2 byte fixed-length format encoding called UCS-2 is used for all tests of the different filters. This encoding support all the major alphabets in the world but the data is assumed to mainly contain characters of the latin alphabet.

Before any data is fed to a filter a pre-processing is performed which consists of a tokenization part and optionally n-gram construction and stemming. The tokenization splits a message into smaller parts called tokens. The splits are done on defined characters or patterns resulting in several substrings when the tokenization has finished.

The n-gram process creates new tokens by stepping through the existing sequence of tokens and for each step combining the current token with the N tokens ahead of it. This process creates a bigger word space which can give more information from a message. While the n-gram can be of any size, too large and the word space becomes unmanageable.

Stemming is in someways the opposite of the n-gram process. It tries to decrease the word space by only keeping the stem of a word. The idea is that a word space too large can be difficult to train a classifier on for several reasons. If a word space is too large it can be difficult for a filter to represent a big enough part of the word space. A larger word space also means that more training data is necessary.

Tokenization for messages in this work uses only the most typical delimiters used in written text, including whitespace, interrogation point and period.

In half of the tests stemming of the messages was done, adapted for the english language. It was used to shrink the word space at the cost of a loss of information for reasons of testing if this would improve the performance for a reasonably large number of features.

There were also tests with different sized n-grams of features to increase the word space while also increasing the available information in each message to test how this would impact on the results of the filtering.

It was decided to use information gain in the feature selection step to rank the features and use the ones most highly ranked for the specific domain for reason that this method seems to be in common use in spam filtering when deciding on features for the feature vector. Information gain is also used in the C4.5 algorithm during construction of its decision tree.

Chapter 3

Learning Algorithms

This chapter gives an overview of each of the machine learning algorithms by discussing firstly how they are trained with the help of training data and lastly how messages are classified. The classifiers are discussed in the order naïve Bayes, decision tree, support vector machine and dynamic Markov coding.

3.1 Naïve Bayes

The naïve Bayes algorithm applied to spam filtering was first brought up in 1998 and led to the development and practical use of many other machine learning algorithms [10]. The implementation of a naïve Bayes machine learning algorithm is very simple, can be quite computationally effective and show reasonably high prediction accuracy compared to its simplicity. Although it is now outperformed by many newer approaches, researchers commonly use it as a baseline to other algorithms and it is one of the reasons to why I chose to use it in my evaluation.

The naïve Bayes formula is based on the Bayes theorem but with an assumed conditional independence. It means that the features are completely unrelated to each other when calculating the conditional probability for them. While this assumption is generally not correct, it has been shown that classification using this method often performs well. It was decided to use multinomial naïve Bayes classifier since it seemed designed for text document classification and take word counts into account which fits a numeral feature vector that is used in this study.

The formula for Bayes theorem is:

$$P(C|X) = P(C) * P(X|C)/P(X)$$

Where X is our feature vector and C is the expected class. Since the denominator will be constant we are only interested in the numerator. We can

use the following formula.

$$P(C_j|X) = P(x_1, x_2, x_3, ..., x_n|C_j)xP(C_j)$$

In this case, x_n are features of the feature vector where n is the feature number from one up to the feature vector's dimension size and C_j is the class type (for example spam or legitimate). By calculating the posterior probability for C_j given X and knowing that each feature of X is assumed to be conditionally independent for naïve Bayes we can rewrite the formula

$$P(x_1, x_2, x_3, ..., x_n|C_j)$$

into

$$H(X) = \prod_{k=1} nP(x_k|C_j)$$

and the formula will be calculated in the following form where the C_j with the highest posterior probability will be the one labelled to X.

$$P(C_j|X) = P(C_j) * \prod_{k=1} nP(x_k|C_j)$$

Knowing this we can see that there are two type of parameters needed to be found during training, the class probability and the conditional probability for each feature given a class.

3.1.1 Training

To construct the classifier, we first need to get each feature from the training data processed to get the parameter estimates for it. This process estimates how likely it is for a certain feature to be found in a message for a certain class C_j relatively to other features. The parameters for the features are calculated as:

$$P(t|c) = (T_{ct} + 1)/((\sum_{t' \epsilon V} T_{ct'}) + |V|)$$

T_{ct} is the total number of occurrences in class C of feature t and $\sum_{t \epsilon V} T_{ct}$ is the total number of tokens found in the documents of class c. 1 in the numerator and $|V|$ in the denominator are for smoothing which will prevent any probabilities to become zero, where $|V|$ is the total number of features existing [15].

We will assume that we are working with numeral feature vectors so the feature count can go from 0 ... n where n is a positive number. Let us assume we have the training data from table 3.1.

Class	Words
Spam	Buy tickets today
Spam	tickets tickets tickets
Spam	You won
Legitimate	Have you got the tickets
Legitimate	Where are you now

Table 3.1: This table shows an example what of words documents in a training data set may contain, and the document class

With the following documents as training data we can now calculate the class probability and the conditional probability of a feature.

Seeing from the table there are three documents of type spam and two documents of type legitimate. The probability the spam class is $P(Spam) = \frac{3}{5}$ and for the legitimate it is $P(Legitimate) = \frac{2}{5}$.

And now for the feature we will use the token *tickets* as an example. We first count the total word count of *ticket* found in the documents labeled as spam which is 4.

$$P(tickets|Spam) = \frac{4+1}{\sum_{t' \epsilon V} T_{ct'} + |V|}$$

Then we count the total number of tokens in the class 'Spam' which is 8.

$$P(tickets|Spam) = \frac{4+1}{8+|V|}$$

And lastly we count the total number of features that we are using, which in this example would be 11 (*buy, tickets, today, you, won, have, got, the, where, are, now*).

$$P(tickets|Spam) = \frac{4+1}{8+11} = \frac{5}{19}$$

The same will be done for $P(tickets|Legitimate)$ which is:

$$P(tickets|Legitimate) = \frac{1+1}{9+11} = \frac{2}{20}$$

Class	Words
unknown	where are the tickets

Table 3.2: An example message to be classified.

Probability	Value
P(Spam) =	$\frac{3}{5}$
P(Legitimate) =	$\frac{2}{5}$
P(where|Spam) =	$\frac{1}{19}$
P(are|Spam) =	$\frac{1}{19}$
P(the|Spam) =	$\frac{1}{19}$
P(tickets|Spam) =	$\frac{5}{19}$
P(where|Legitimate) =	$\frac{2}{20}$
P(are—Legitimate) =	$\frac{2}{20}$
P(the—Legitimate) =	$\frac{2}{20}$
P(tickets|Legitimate) =	$\frac{2}{20}$

Table 3.3: Results for each feature.

This calculation will be done for all features available over the whole training set. The results are what is later used in the classification part to find the probability for a message to be either legitimate or spam.

3.1.2 Classification

Using table 3.1 as an example, an example of how a possible classification would be done will be explained. Let us assume we want to classify the message:

The values for the necessary parameters are as seen in table 3.3

When we have all the values we want to calculate the probability for the document to belong to spam class and the probability for it to belong to the legitimate class.

$$\begin{aligned} P(Spam|unknown) &= P(Spam) * P(where|Spam) * P(are|Spam) \\ &*P(the|Spam) * P(tickets|Spam) \\ &= \frac{3}{5} * \frac{1}{19} * \frac{1}{19} * \frac{1}{19} * \frac{5}{19} \\ &= 2,3 * 10^{-5} \end{aligned}$$

$$\begin{aligned} P(Legitimate|unknown) &= P(Legitimate) * P(where|Legitimate) * P(are|Legitimate) \\ &*P(the|Legitimate) * P(tickets|Legitimate) \\ &= \frac{2}{5} * \frac{2}{20} * \frac{2}{20} * \frac{2}{20} * \frac{2}{20} \\ &= 4,0 * 10^{-5} \end{aligned}$$

Seeing from the results in table 3.3 that $P(Legitimate|unknown)$ is larger than $P(Spam|unknown)$ the classifier would in this case have classified our new message as a legitimate message and not as spam.

While the conditional independence assumption makes calculation of the posterior probability of each feature conceptually simple, computationally efficient, and in need of little training data because of a small amount of parameters. The draw back is also caused by this, since no information are taken into account how different words might relate to each other. If in examples where this assumptions proves mostly correct this would of course not be a drawback.

3.2 C4.5 Decision Tree Learning

The decision tree learning algorithm used in this study is the C4.5 algorithm. This algorithm is an extension of an earlier algorithm called ID3 with improvements such as support for continuous feature values [18]. As all the other algorithms in this study it first needs training data to train it. With the training data it starts from the root node and recursively splits it on the most appropriate feature it can find by the use of some feature selection technique [4]. When splitting occurs a decision node is created which controls what sub-branch to choose at this point when an incoming message is being classified. The decision node remembers what feature the training data split on and what the feature values is needed for each of the branches.

After recursively splitting the data it will eventually arrive at a node where splitting the training data again either is not possible or where there is no more decrease in the information entropy of the training data. In this case a leaf is created and labeled by the majority class in the current training data.

When the creation of the decision tree is done, it is typically but optionally, pruned to decrease the size and preferably give improvement in the trees performance to classify future data. This is done through a heuristic test called Reduced-error pruning [18] which estimates the error of a node compared to its branches to decide if it should be replaced by a leaf.

3.2.1 Training

To start construction of a C4.5 decision tree, a training data set of classified feature vectors such as $C = c_1, c_2, .., c_m$ where $c_1, c_2, ..., c_m$ represents different classes as to which the training data can be classified in.

The construction starts at the root node with the training data set T to use for construction. It begins by recursively doing several checks based on Hunt's Method [18].

1. T only contains one type of class, the tree becomes a leaf and is assigned to the class of the data set.

2. T contains no cases, the current tree becomes a leaf and C4.5 decides which class the leaf should be associated with by finding the majority class of the trees parent.

3. T contains a mixture of classes and should be tried to split on a single feature with the purpose that each subset is to be refined so that they move closer to having only one class in its collection of cases. The feature have one or more mutually exclusive events with outcomes $O_1, O_2, ..., O_n$ giving the subsets $T_1, T_2, ..., T_n$. The current tree node will become a decision node based on the feature chosen. The outcomes will be the n branches and they are recursively processed. The ith branch having the outcome O_i will construct the subtree with the training data T_i.

Splitting

To split the training data set on the most appropriate feature, it is chosen based on testing the information gain or information gain ratio of each possible feature in the feature vector. The feature with the highest information gain ratio is then used to split T on, although each subset of T can not be too small in its number of cases. The minimum number of cases can vary but the default value in C4.5 is 2.[10] If any of the subsets is below the minimum number, no split will occur and a leaf node will be created instead of a decision node which stops the recursion on this branch.

In C4.5 there are three different kinds of splits that can be performed.

(a) Split on a discrete feature which for each outcome will produce a branch

(b) Split similar to first test but where different outcomes may be grouped together. Instead of having one branch for one outcome, several outcomes may share the same branch.

(c) Split on an feature with continuous numeric values. This is a binary split, two outcomes. To split T on a continuous feature f with a feature value A, the conditions should be such that $A <= Z$ or $A > Z$ where Z is a value for a possible split.

The split used in this study is split c) and information gain ratio is used as comparator for the split. As an example of how the information gain ratio from a split on a feature would be calculated, I will denote by $|T|$ the total number of cases, T_j represents a possible subset after a split on T by a decision node and $freq(c_i, T_i)$ represents the frequency of a class c_i in the subset T_i. Lastly $proportion(c_i, T_i)$ will denote $freq(c_i, T_i)/(|T|)$.

The information gain ratio is the chosen comparator for which feature to split the training data on. Information gain shows a bias to splits with many outcomes while information gain ratio solves this problem. Information gain and entropy which is relevant here is discussed in section 2.1.5.

given the formula $IG(T, a) = H(T) - H(T|a)$ for information gain, let us us assume there are two features of interest which result in the following two trees.

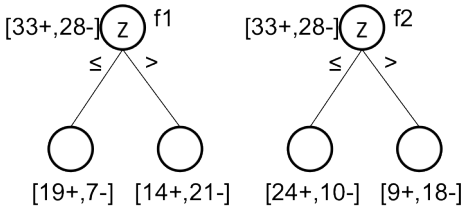

Figure 3.1: Example of two different splits for a decision node on either feature 1(f1) or feature 2(f2) while using the same training data.

In the figure the positive class is denoted by c_1 and the negative is c_2. To find which of these splits creates the best result we firstly calculate the entropy for if there would be no split, that is to say if a leaf would be created instead. We can see from the figure that $|T| = 61$, $freq(c_1, T) = 33$ and $freq(c_2, T) = 28$. Then knowing this the current entropy can be calculated.

$$\begin{aligned} H(T) &= -proportion(c_1, T) * log_2(proportion(c_1, T)) \\ &\quad -proportion(c_2, T) * log_2(proportion(c_2, T)) \\ &= -(\frac{33}{61}) * log_2(\frac{33}{61}) - (\frac{28}{61}) * log_2(\frac{28}{61}) \\ &\approx 0,995 \end{aligned}$$

Now when the current entropy is known, the entropy for choosing one of the splits should be sought after. The feature $f1$ will be chosen in this example using the equation $H(T|\alpha)$ where α is the chosen feature.

$$\begin{aligned} H(T|f1) = \sum_{i=1} & 2H(T_i) - proportion(c_1, T_i) * log_2(proportion(c_1, T_i)) \\ & -proportion(c_2, T_i) * log_2(proportion(c_2, T_i)) \end{aligned}$$

Here T_i are the two data sets created by a split on feature $f1$ giving the entropy for set 1:

$$H(T_1) = -\frac{19}{26} * log_2(\frac{19}{26}) - \frac{7}{26} * log_2(\frac{7}{26}) \approx 0.84$$

The entropy for set 2:

$$H(T_1) = -\frac{20}{35} * log_2(\frac{20}{35}) - \frac{15}{35} * log_2(\frac{15}{35}) \approx 0.98$$

The information gain is then found by the equation:

$$IGT(t, \alpha) = 0.24 - H(T|f1) = 0.995 - \frac{26}{61} * 0.84 - \frac{35}{61} * 0.98 \approx 0.075$$

So by choosing feature $f1$ the information gain is 0.075. Doing exactly the same steps for $f2$ we get an information gain of 0.10. To find the information gain ratio we divide the information gain by the potential split's intrinsic value.

Using the formula:

IV(T,a)=- $\sum_{v \epsilon values(a)} \frac{|\{x \epsilon T | value(x,a)=v\}|}{|T|} * log2(\frac{|\{x \epsilon T | value(x,a)=v\}}{|T|})$

The intrinsic value for the first split is ≈ 0.984 giving us a gain ratio of 0.076 for $f1$ while the gain ratio for $f2$ turns out to be 0.101. In this case it turned out to be no real difference from the information gain. Seeing as how $f2$ gives the highest information gain ratio, this is the feature that should be chosen. The training data will be split into two sets and each set will continue building a new subtree. On the other hand if there would have been no information gain to be had or if the average information gain from all possible splits would have been higher that the current split's, the current node would become a leaf. I would be given a class distribution of each class' probability based on the training data set distribution. The recursion would then stop for this branch.

Pruning

When construction of the tree is done it may optionally be pruned to combat over-fitting using a method called reduced-error pruning. Over-fitting occurs when the tree does not generalize well such that its a higher classification error rate on the test data on the training data. The tree being too complex can cause this. The noise in the data could then have more of an effect on the leaves; it could also be because of for example too little training data.

The construction starts at the root node. There are two different methods of pruning offered in C4.5; subtree replacement and subtree raising. In the implementation the later one is optional if pruning is done.

Pruning is done in a left to right, top to bottom fashion where the decision nodes nearest the leaves are compared for pruning and then recursively working downwards to the root of the tree. The aim is to find decision nodes or a subtree from one of the decision nodes, that has a theoretically lower classification error rate, and if so found then replace the current node with a leaf, or to the compared subtree. Through this process the average error rate on the training data should decrease and hopefully make the tree First subtree replacement is examined, and if no pruning is done, optionally subtree raising is examined.

The error estimate is given by calculating the upper confidence interval $U_{cf}(E, N)$ for the binomial distribution of the leaf for a set confidence level (default is 25%). Here E is the number of incorrectly classified training examples in this node, and N the total number of training examples. Given the total number of examples and the total number of errors in the leaf (the sum of the minority classes) the error estimate is given by $N * U_{cf}(E, N)$, multiplying the upper confidence interval by the number of total cases in the leaf. The confidence level for is used as a tool for how hard to prune a tree, and the higher the confidence level the less amount of pruning is done. The calculation of the error is based on the already existing distribution from the training data, thus no extra data is used to compute errors in the tree.

Subtree replacement is when a decision node is found to have a theoretically lower classification error rate; would it be a leaf; than its branches' weighted sum of error. If that is the case a leaf would replace the decision node and a class distribution giving each class probability will be created for the leaf.

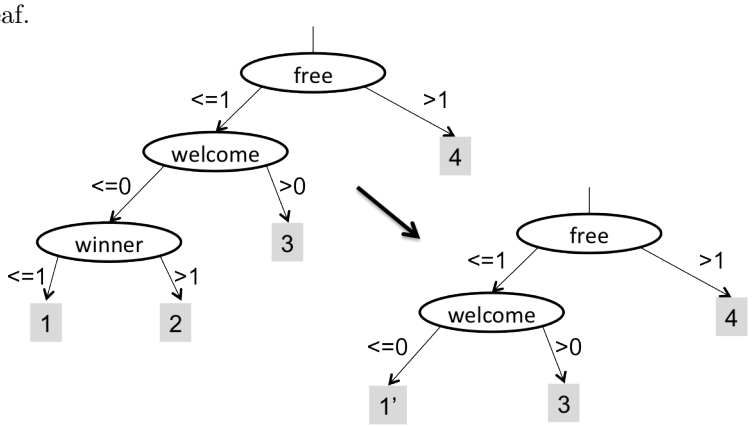

Figure 3.2: Example of how a subtree replacement is done.

In this example it is found that the decision node with the feature *winner* has a lower estimated error than its branches. Therefor a subtree replacement is done. The decision node is remade into a leaf and the probability distribution from its earlier branches constitutes the leaf. *1'* in the figure is the new probability distribution for the leaf.

Subtree raising on the other hand will compare a decision nodes biggest branch's error estimate to the error estimate of the tree starting from the node. If the tree of the biggest branch has a lower error estimate the tree's root node will replace the parent node and the training data the smaller branch will be redistributed to the larger branch's nodes. The effects of subtree raising is said to give ambiguous results, in some cases it may improve the precision of the classifier [19].

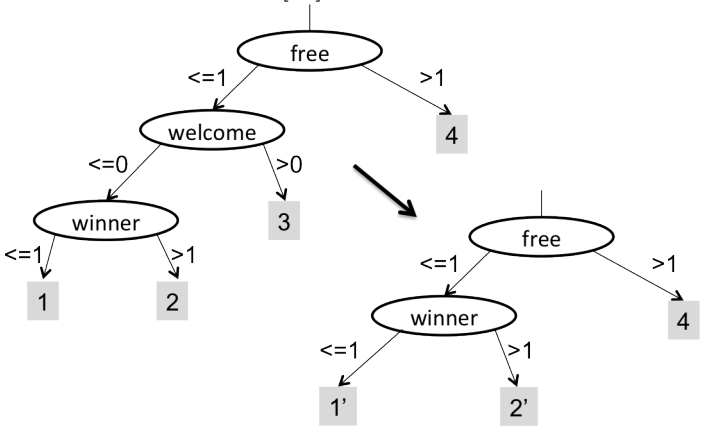

Figure 3.3: Example of how a subtree raising is done.

Here *winner* is shown is shown to have a lower error estimate than what the decision node with the feature *welcome* has. The decision node *winner* and its branches are moved to replace the *welcome* node. The training cases from the former right branch is being redistributed to *winners* left and right branches. The new distributions in this example are *1'* and *2'*.

3.2.2 Classification

Classification is done by beginning at the root node and going down one of the branches in the current decision node until a leaf is reached. The choice of branch taken when in a decision node is based on which feature the decision node decides on and what the threshold is for the feature. If a feature in the message being classified has a feature count of equal or lower for the specific feature chosen by the decision node, the left branch will be chosen, otherwise the right branch is chosen.

As an example assume that a message with the following feature vector seen in table 3.4 is chosen.

Feature	Value
free	1
thanks	0
charge	1
welcome	0
winner	2

Table 3.4: This table shows a feature vector.

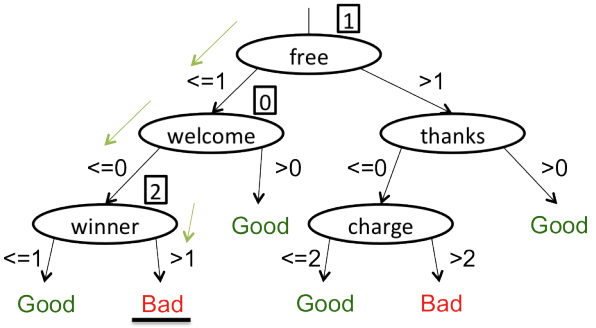

Figure 3.4: Example of a a clasification process for a decision tree.

In the example there are only natural numbers since in the spam filtering context there are only counts of words. The root node in this decision tree decides on the feature *free* and its threshold is less or equal than 1 for the left branch and larger than 1 for the right one . The example feature vector only has a value of 1 for this, which means the left branch is taken. In the next node the feature vector once again takes the left branch. For the last decision node for this branch the value for the feature *winner* exceeds the threshold meaning that the right branch is finally taken. A leaf node is reached and a decision is given. In the example a good result means that it is a legitimate incoming message and a bad result means it is spam. In this case the message was classified as spam and will be blocked.

When a leaf in the tree has been reached, it returns what class label C it has along with its probability. The probability is given by the ratio of $\frac{K}{N}$. K is here the number of training examples in this node from class C and N are the total number of training examples which reached this node.

3.3 Support Vector Machines

The support vector machine, or SVM are today seen as one of the best off-the-shelf machine learning algorithms. The main idea of this classifier is to treat each feature vector as a point in a high-dimensional space, where the size of the space is controlled by a kernel function. In text classification

a simple linear kernel is often used due to the space already being large, resulting in an n-dimensional space where n is the number of features in the feature vectors. The goal is to find a hyper plane in this space that can separate any point given feature vectors into two different classes, such as spam and legitimate messages. The hyper plane should not only separate the points in space though, but the goal is to find the hyper plane which have the largest possible margin from all points in the training set.

3.3.1 Training

Assume that we have two classes, where $y \in \{1, -1\}$ is a class label and $x \in R^n$ is a vector built from the training data as discussed in section 2, with n being the dimension of the feature vectors. Thus we have a training set of pairs (x_1, y_1) to (x_n, y_n) where n is the number of training cases.

Now a hyper plane $w \cdot x - b = 0$ would be sought that has as large geometric margin as possible to the nearest points and where the constraint is that for each case $y_i(w \cdot x_i - b) \geq 1$ for all i. Here w is the normal vector of the plane. The geometric margin is twice the size of the margin to the nearest point from the hyperplane.

Now a hyper plane $w \cdot x - b = 0$ would be sought that completely separates the two classes where the hyperplane is as far away as possible from each of the nearest points for each class. To find this hyperplane let us assume we have two hyperplanes $y_i(w \cdot x_i - b) = 1$ and $y_i(w \cdot x_i - b) = -1$ as seen in figure 3.5 which both separates the data and do not overlap. As can also be seen the margin between these two planes is defined as $2/(||w||)$.

To find a plane which separates the two classes the best the margin between these two planes should be as large as possible meaning w should be small. The constraint $w \cdot x - b \geq 1$ for $y_i = +1$ and $w \cdot x - b \leq 1$ for $y_i = -1$ is added to stop any data point to appear in the margin. The first constraint would be applied to the class which in figure 3.5 is separated by $y_i(w \cdot x_i - b) = 1$ and the second constraint for $y_i(w \cdot x_i - b) = -1$. To simplify it these two constraints can be written as $y_i(w \cdot x_i - b) \geq 1$ for all $1 \leq i \leq n$.

We can formulate this problem into a constrained optimization problem with the objective $\underset{w,b}{\text{Minimize}} ||w||$. This is subject to the constraint $y_i(w \cdot x_i - b) \geq 1$ for i from 1 to n mentioned earlier [3].

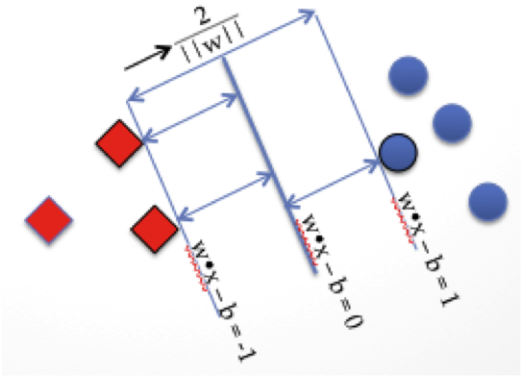

Figure 3.5: During training, the only cases that have an effect on the hyper plane will be the ones with a point just on the margin, called the support vectors. The two different shapes represents two different classes.

From figure 3.5 we can see that the smaller the norm of w becomes the larger the margin and thus w should be minimized to find the hyper plane which best separates the different classes. The norm of w contains a square root, but it can be removed without changing the result to simplify the calculations and $1/2$ is multiplied as well for the same reason. The objective thus becomes $\underset{w,b}{\text{Minimize}}(\frac{1}{2}) * w^2$.

Often it is not possible to find a hyper plane which can completely split the two classes. Instead the cases that prohibit a hyper plane from splitting the two classes must be taken into account when trying to minimize the margin. A solution for this is the so called Soft Margin method which allows mislabeled classes. When using this method, the earlier constraint will be modified to

$$y_i(w \cdot x_i - b) \geq 1 - \nu_i$$

where ν_i is the error for how far inside of the margin a point is. The objective changes as well into

$$\underset{w,\nu,b}{\text{Minimize}}\{(1/2) * w^2 + C * \sum_{i=1} n\nu_i\}$$

to take into account this error [7]. This is called the soft-margin sum and here C is a constant called the regularization parameter that controls how much the penalty of the errors that should be taken into account. The size of the penalty will be on the cost of the margin size.

When a optimum has been found a hyperplane $w \cdot x - b = 0$ is given and classification on the trained classifier can now be done.

3.3.2 Classification

Classification for a linear SVM is a simple task. Since the hyper plane is found and the incoming message should have been transformed into a feature vector, the margin between the hyper plane and the feature vector is calculated. $w \cdot x - b$ Here the feature vector is x and the value of the result decides how the incoming message will be labelled. If the value is less than zero it will be labelled as one class, and if the value is larger than zero it will be labelled as the other. The threshold value of zero can of course just as the other classifiers be varied to control how strict classification should be done to decrease the false positive rate or increase the true positive one during classification.

3.4 Dynamic Markov Coding

Dynamic Markov coding (DMC) comes from the field of statistical data compression but just like other compression algorithms like prediction by partial matching(PPM) it has been shown to perform well for classification. What distinguishes the DMC algorithm from the other filters in this thesis is that DMC does not require any pre-processing of messages, instead a message is modeled as a sequence.

> "*By modeling messages as sequences, tokenization and other error-prone preprocessing steps are omitted altogether, resulting in a method that is very robust*" [2]

The three previously mentioned filters all rely on some type of pre-processing and at least in this thesis tokenization of messages to construct the feature vectors to represent each message in a bag-of-words approach. The DMC algorithm on the other hand directly process a message as a stream of bits when classifying what type of class a message might be. By using bit streams instead of tokenizing messages, DMC can avoid the drawbacks that tokenization can incur as mentioned in the earlier pre-processing topic 2.1.4. By having no pre-processing the filter also saves time by not needing to tokenize an incoming message, which can take up a big part of the filtering time.

> *The basic idea of using data compression in classification and other machine learning tasks has been reinvented many times. The intuition arises from the principal observation that compact representations of objects are only possible after some recurring patterns or statistical regularities are detected [2].*

The intuition to use a compression algorithm for a classification task is that just as classifiers they try to find recurring patterns. While algorithms

in data compression search for distinctive patterns in a data to create a smaller representation, a classifier similarly finds the patterns to classify data. Statistical data compression algorithms create a statistical model of the source to be compressed. This model gives the probability distribution of the source, which in turn can give a estimated probability of the source. Having this information the algorithm can then find proper codes to compress the source as, but for classification this last step is omitted and the statistical model is instead used to compare how well an incoming message compares to the model.

3.4.1 Training

The training part of a DMC classifier is the construction of the statistical models mentioned previously. Each model is constructed as a finite state machine and every state has a probability distribution over which state to transition to. The probability distribution is used to predict which binary digit is in the next step of the source.

The model can start out in any kind of predefined state such as a model containing a single state. Another possibility is to start out with a order-7 binary Markov model where the next state depends on the previous 7 states. This starting model is used in the study and it is custom for byte-aligned data. It has been shown that starting out with a larger model like this can improve the results of training the model.

When building a model, a collection of training messages belonging to the certain class which the model is supposed to represent is read. These messages are read as a stream of bits. It is read bit by bit and updates the current states transition frequency count by one while transitioning through them from the current state to the next. When a transition in the model has reached a high frequency count, an expansion process will occur creating new states which gives a more complete statistical model of the training messages.

To show how the construction of a model is done, let us assume that our model started out as a single state and that the training process has run a few steps. Our current model is in figure 3.6 and has already expanded one state. The model shows one output transition going from A to B and one from B to A, otherwise they go back to themselves. The current state is in A.

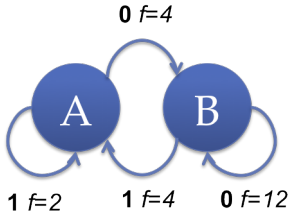

Figure 3.6: Markov model before expansion.

The training data is said to be read as a stream of bits instead of character by character as how PPM does or token by token as how the previous algorithms in the study do. For the example the stream could look something like the following.

...001011100110...

This means that the next state to go to would be state B and the transition from A to B would need to increment the frequency count. For each state transition, a check will always need to be made to see if the model should be expanded. The model is expanded When both the inbound transition has reached a set threshold for the frequency, and the output frequencies of the target state has reached a threshold. If these requirements are met, cloning will be done on state B.

The cloned state B', will have its outbound transitions pointing to the same states as state Bs outbound transitions points to. The outbound transition from A to B will start pointing to the new state B'. The outbound transition frequencies from state B will be redistributed between itself and state B'.The distribution depends on a proportion of the number of times the cloned state was reached from the former state, relative to the outbound frequency count from the target state is [2].

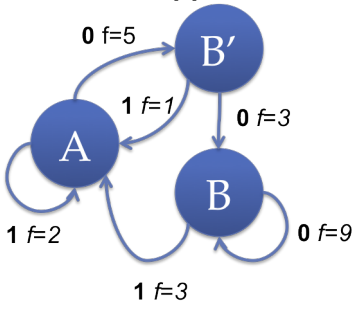

Figure 3.7: Markov model after expansion.

In figure 3.7 we can see that the transition from A to B caused a state cloning operation. State B had before the transition a total outbound frequency count of 16(12+4), so when the state cloning operation occurs there

is 4 times more outbound frequency counts than the inbound frequency from a A to B. This means that the new clone will be assigned a quarter of the frequency counts on its outbound transitions while state B will retain three quarters. Only after the ratio of distribution is calculated is the transition updated, that is why we see the ratio $\frac{4}{16}$ instead of $\frac{5}{16}$.

Through this cloning process of states during the construction, this is how the model continuously grow as it is trained. With more and more data it will increase the number of states in the model and thus making the predictions of the transitions able to depend on deeper context information. That means that the model should be able to return a higher probability during classification on a message that has a pattern that strongly relates to the class of the model.

The training of a model is finished when all the training data belonging to that class has beed processed. The starting state is where each cross-entropy calculation will start from in the given model, is set and there is nothing more to be done for training.

3.4.2 Classification

There are several ways to do classification utilizing the Markov models built, respectively for the case. One of the methods used by Bratko et al. [2] and utilized in this study as well is Minimum cross-entropy (MCE) $H(X, M)$. If doing compression it would tell how many bits in average per symbol the compressed data would be, compared to the original source when using the model M.

$$H(X, M) = E_{x \succ p}(\frac{1}{|x|} * L(x|M))$$

Where x is an incoming message, M is the statical model constructed from training data of one class type and $L(x|M)$ is the shortest possible code length after compression and $|x|$ is the length of the message. The exact cross-entropy would need the source distribution to calculate but

$$H(X, M) \approx \frac{1}{|x|} * L(x|M)$$

is a good approximation if X is long enough and the model M is a good representation for the patterns in X.

> *The cross-entropy ... determines the average number of bits per symbol required to encode messages produced by a source ... when given a model ... for compression [2].*

By comparing the cross-entropy for the source X over each model representing a message class, classification of the source can be done by choosing

the model resulting in the lowest entropy. The intuition is simply that if there is a low cross-entropy, the model would compress the message well. If so it is also likely that it belongs to the class that this model is representing.

To classify a message the cross-entropy for a message for different models is compared and the one with the lowest entropy is generally chosen. However finding the exact cross-entropy requires the source distribution which will not in general be known by the model. The model will instead give an estimate and with a long enough message x it will likely be near the real cross-entropy

The approximation of the cross entropy is:

$$H(X, M) \approx \frac{1}{|x|} * L(x|M)$$

Here $L(x|M)$ is defined as:

$$L(x|M) = -log(\sum_{i=1}(|x|)f(x_i|x_{i-k}^{i-1}, M))$$

Each probability given by f for x_i is dependent only on a limited number of states before, this depends on the order of the Markov model. The larger the model is the more context information and the better the approximation should be. The equation $H(X, M)$ can be rewritten to $H(X, M, d)$ where x is simply substituted by d, and the final equation for classification is:

$$c(d) = \underset{c \epsilon C}{\operatorname{argmin}}(X, M_c, d) =$$
$$= -\frac{1}{|d|} * log(\sum_{i=1}(|d|)f(d_i|d_{i-k}^{i-1}, M_c)).$$

Here $c(d)$ is the equation which, given a document d returns the class type c for the model M_c that achieves the lowest cross-entropy of all models available.

For a simple example of how the extremely short message 'A' would be classified, let us first look at its binary value. The binary value for UTF-8 encoding would be '01000001'. Let us use the model in figure **??**, here we start in state A.

The order in which to read the bits should not be important as long as it is consistent, so reading from left to right the first digit is a 0. This gives us our first probability:

$$\frac{5}{5+2} \approx 0.71$$

The current state has switched to B' and the new digit is now 1 which switch back to state A again which gives us the probability:

$$\frac{1}{1+3} = 0.25$$

Current state is now A again and the following 5 digits are zeros. This means it continues on as the two first step and the only difference is that the state will remain the same. This accumulates on until the end of the message is reached.

$$log(\sum_{i=1} |d| f(d_i|d_{i-k}^{i-1}, M_c)) =$$
$$log(0.71 * 0.25 * 0.71 * 0.75 * 0.75^3 * 0.42)$$
$$\approx log(0.0167)$$
$$\approx 5.89$$

Here M_c is our example model and where d is our message 'A'. If the length for this message is 8, then the cross-entropy would be:
$H(X, M_c, d) = \frac{1}{8} * 5.89 = 0.73625$.

This is how a cross-entropy calculation between a message and a model is done. When classifying the message, the class which model returns the lowest cross-entropy can be chosen. Other methods such as in a two-class example such as done in this work (i.e a spam class and legitimate class) is to make a division between the result from the spam model, and the result from the legitimate model. Thereafter a threshold value can be set which decides if a message should be classified as spam or as legitimate depending on if the value given by the ratio between the two results exceeds the threshold or not. Thus it creates more control and fine-tuning for operation a filter like this.

Chapter 4

Evaluation of Algorithms

After the four machine learning algorithms had been chosen, analysis on each of them were done to see how viable it was to use them for filtering SMS-data. The most important of the requirements to follow would first and foremost be to keep the low false positive rate that was asked for as mentioned in section 1.1. Second most important requirement would be for the filter to be fast enough. Since the space limitation was rather generous I did not expect it to break the limit. Training of filters is not expected to be run often enough to justify discarding a classifier because of it. The testing on the naïve Bayes, C4.5 decision tree and the SVM algorithms were done using the WEKA testing environment whilst a free implementation of DMC were not found and therefor was created. The DMC implementation was done in Java.

4.1 System

Testing was done on a standard laptop computer with a 2GHz Intel Core 2 Duo and 1 GB of RAM.

4.2 Settings

A ten-fold cross validation as mentioned in section 1.6.3 was performed for the evaluation of each configuration of each filter. The results of these evaluations were then averaged to get a good estimate of each filters performance.

The filters were tested with a feature vector size of 500, 1000, 1500 and 2500 words.

Each test configuration only used a lower-case representation of characters; this resulted in a smaller word space since capital characters were not represented. While losing some information by not representing capital characters, it also meant that the size of the training data might not need

to be as large. By removing this information from the data it can possibly decrease performance in the classification accuracy but it was argued that by using a smaller word space a not as big of a feature vector size would be necessary. Also not as many messages would need to be classified since a smaller amount of training data should be needed because of the smaller word space.

Stemming was used in half of the tests to see if there was any improvement. Once again the same reasoning was used, with less permutations of each word, with stemming on, the word space would be smaller and some information would be lost. The same number of features in the feature vectors could instead be used to represent more words than in a bigger word space with stemming off.

N-grams in the size of unigrams and bigrams of tokens were used to represent features in the feature vector. The tests vary between unigrams and unigrams coupled with bigrams. By using bigrams the word space will become expanded instead and more information will be available, it was argued that there could be more distinct information to get between correlation of tokens than by lowercase and uppercase characters. So the decision was based on hopefully finding pair of tokens that were especially distinctive to either spam or legitimate messages.

4.3 Data

To analyze the performance of the algorithms, data for training and testing them were necessary. Generally the more data, the better the classification accuracy should be up to a point. To get a convincing result real communication data was used for the test. Data for training and testing them was not classified if it was a spam or a legitimate message and this had to be done manually. A total number of 14007 messages were classified, where 8141 were classified as legitimate messages while 5866 messages were classified as spam.

4.4 Speed and Memory Consumption

Comparing the difference in precision, recall, accuracy are important metrics for examining how correctly classification is done. However knowing build times of a filter, memory consumption and most importantly, classification speeds are also very relevant. An SMS-server may need to handle many thousands of new incoming messages per second and in such an environment the filter cannot become the bottle-neck for the operation.

The filters using a naïve Bayes, C4.5 decision tree or SVM classifier does not only consume time for the classification stage. Each assume the messages to be represented as bag-of-words which require them to be pre-processed. In the first part of this analysis the time needed for training is presented,

Using stemmer:	False	True
Feature selection time:	16.5 min	15 min

Table 4.1: This table shows the feature selection times with unigrams.

Using stemmer:	False	True
Feature selection time:	78 min	73 min

Table 4.2: This table shows the feature selection times with unigrams + bigrams.

here DMC is left out since it does not use feature selection nor the bag-of-words presentation. This includes feature selection and classifier training. The second part goes through the time consumption when classification is done. Feature vector construction and classification time is mentioned here. Lastly the memory usage is presented for each of the classifiers using different setups.

4.4.1 Time Consumption (Training)

Classifiers that are depending on the bag-of-words representation which in these experiments are naïve Bayes, C4,5 and SVM require feature selection. Feature selection is done firstly before training the classifier to limit the scope and find the relevant features.

Tokenization of the training messages was only a fraction of the total training time. The tokenization part varied between 5-15 seconds depending on if stemmers were used, and if bigrams were included or not. Because of this small difference, it was deemed unimportant to take it into consideration.

The result from table 4.1 and 4.2 shows that with the hugely increased dimension from adding bigrams, the time consumption also increases greatly. If a fast construction time would be important this likely would cause a problem for larger sets of training data. In this domain the filters are not expected to update with such frequent rate that this would case a problem.

DMC was given a separate average 'training time'-table, since it does not share the same underlying architecture of using a bag-of-words representation for messages like the other three classifier. Big and small represents the big and small threshold values used during construction of the DMC classifier for controlling how fast the Markov-chain model should expand.

DMC has by far the fastest training time of all classifiers evaluated as seen in table 4.3 and 4.4. The small difference in construction times for the different setups of DMC should be related to the fact that the lower threshold setups will expand nodes faster, and thus spending more time constructing new nodes compared to when using a higher threshold setting. The implementation made for the test is also far from optimal, so it is likely

# of features:	500	1000	1500	2500
Naïve Bayes	1 sec	1 sec	1 sec	1 sec
C4.5	6.5 min	16.5 min	30 min	48 min
SVM	4.5 min	6 min	7 min	7 min

Table 4.3: This table shows the average training time for naïve Bayes, C4.5 decision tree and SVM.

Big/Small:	2/2	4/4	5/5	6/6
DMC	4.5 sec	3.5 sec	4 sec	1.9 sec

Table 4.4: This table shows the average training time for DMC

that the implementation is also a bit at fault. Interestingly it is by far the fastest, even when the time for feature selection for the other classifiers is not included.

Naïve Bayes time complexity is $O(pN)$ where N is the number of training cases and p the number of features. It is not apparent in the tests but it might be just too few tests, since the times recorded would vary with up to twenty seconds between different folds during cross-validation. Still it has a several times faster construction time speed compared with C4.5 decision tree and SVM.

C4.5 has a time complexity of $O(p * N * log(N))$ for constructing the tree and a time complexity of $O(N)$ for subtree replacement which was used for pruning. The time differences for increasing number of features become quite large when the feature number reaches 2500 and is by far the slowest to train.

SVM using the SMO algorithm to solve the optimization problem has at least a time complexity of $O(N * L)$ where L is the final number of support vectors [5]. It is difficult to say something about the time increase for increasing the number of features. It only increases time by a small margin when increasing the number, as it should considering its time complexity. It is much faster than C4.5 in time consumption for large number of features, but it does not come close to the speeds of the DMC or naïve Bayes classifiers.

4.4.2 Time Consumption(Classification)

Construction time however, is not the only thing to take into account. What is more pressing is the classification times for each classifier. Classification time depends on two factors, namely pre-processing and the classifier itself. Pre-processing tokenize a message and optionally stemming and building n-grams is included here. When this step is done the feature vector is constructed from the information extracted. After this step the classifier comes into play and uses the newly built feature vector to do a classification

Using stemmer:	False	True
500 features	9500	5800
1000 features	6800	4750
1500 features	5100	4000
2500 features	3800	3000

Table 4.5: This table shows the average number of messages, per second, being tokenized and having feature vectors constructed for with unigrams.

Using stemmer:	False	True
500 features	5000	2800
1000 features	4300	2300
1500 features	3200	2200
2500 features	2800	1900

Table 4.6: This table shows the average number of messages, per second, being tokenized and having feature vectors constructed for with unigrams + bigrams.

of a message. As mentioned in section 1.1 the number of messages to be possibly to filter per second should be at least around 10.000. Taking into account however that the experiments were done on different hardware, a bit less than this was acceptable as mentioned in section 1.4.

Having a larger number of features means that there will be a large number of indexes in the feature vector for each token to compare with to know if it have a place in the feature vector model. The time increase for using more features gradually decreases. Where for example in table 4.5 the difference in messages per second from 500 features to 1000 is 2700 messages per second. The difference from 1000 features to 1500 is only 1700. Such a curve is a sign of the use of a binary search with time complexity $O(log(n))$ where n in this case is the number of features in the feature vector. This is what is used by the chosen feature vector format in WEKA. A more efficient data structure like a hashmap might be preferable.

Obviously including the bigrams would slow feature vector construction down even further. Not only will it take a bit of time to make bigrams out of the tokens (k-1 bigrams where k is the number of tokens in a message), but there will also need to be more searches to the feature vector model. Stemming might lessen the number of searches being done to against the feature vector model, but the stemming processing itself is obviously also costly since it needs to compare a word root's possible suffix and thus again some kind of search will be implemented.

Using both bigrams, stemming and having a high number of features affects the speed for constructing feature vectors greatly. If accuracy in classification does not suffer by excluding one or more of them, it would be

# of features:	500	1000	1500	2500
Naïve Bayes	315000	270000	220000	175000
C4.5	87000	73000	70000	66000
SVM	63000	40000	26000	22000

Table 4.7: This table shows the average number of messages per second classified by the bag-of-words dependent classifiers.

Big / Small:	2/2	4/4	5/5	6/6
DMC	14000	14000	14000	14000

Table 4.8: This table shows the average number of messages by per second classified by the DMC classifier.

beneficial to do so. There is a possible trade-off between accuracy and speed in the application domain. It is important that the speed of the filtering system is fast enough to be able to filter all incoming messages, but it can not be allowed to affect the accuracy rate to a high degree. If the filter will end up having a too high proportion of false positives because of cutting down on one of these options, then it is not a viable solution to solving possible speed issues.

The fast classification speed for the naïve Bayes classifier shows that there is no discussion about it that this classifier will be fast enough for the speed requirements stated in section 1.1. Here all of the other results are very acceptable as well, while again the results from the classifier tables do not include the time it takes to prepare a feature vector for their classification process. DMC is not dependent on pre-processing, and is thus much faster in total to classify messages. With pre-processing included which is definitely the bottle-neck many of the results are acceptable, though as can be seen from table 4.6 there are some very low speeds especially when using 2.500 features. A pre-processing which can at most handle around 3.000-4.000 messages per second will most likely not be fast enough to filter enough messages per second on the goal hardware.

As can be seen the speed for classifying messages for the DMC classifier is constant. This is as it should with the time complexity of the DMC classification task being linear in the length of the testing data.

4.4.3 Memory Consumption

Even though memory is quite abundant and the limitation of 1 GB of RAM is not expected to be breached, it were still necessary to inspect. The first three classifiers dependent on the bag-of-words representation was all below 5MB of memory usage. While the naïve Bayes and C4.5 decision tree classifiers never reached above 1 MB of memory usage in any setup, SVM topped at near 5 MB. DMC however with its different approach to handling messages

Big / Small:	2/2	4/4	5/5	6/6
DMC	180MB	118MB	84MB	77MB

Table 4.9: This table shows the size of the DMC classifier's Markov model for different settings.

by directly inputing their bit-streams into a Markov chain model fared a bit differently.

A problem with the DMC classifier is that there are no methods like feature selection or pruning to decrease the amount of data taken into account while training the classifier. As more training data are used, the memory consumption of the classifier will also increase. It would not be unexpected if in a real situation more training data would be used. The would have to be several times larger than the amount used here though to possibly break the memory limit of 1 GB of RAM as is mentioned in section 1.1.

There are solutions though to the continued growth of the Markov model. By keeping a count of how many nodes that have been created the thresholds can be set to increase step by step as the node count increases, to slow down the expansion of the model. This should also mean though that the rate of learning decreases.

4.5 Classification Accuracy

Even though the speed of the filter has to be fast enough, the accuracy has to be high as well to catch enough spam. The precision has to be high as well since catching many legitimate messages would turn into bad business and unhappy customers. The classifiers will be presented by firstly naïve Bayes, then C4.5 decision tree, SVM and lastly DMC.

The diagrams shown are the ROC-curves from the results of classifying training data. It has been limited in x-axis to 0-10% false positive rate, while the y-axis has been limited to 80-100% of true positive rate to highlight the more interesting and relevant area of the filters.

4.5.1 Naïve Bayes

Naïve Bayes is the simplest of the filters evaluated. As mentioned in section 3.1 each feature from the feature vectors used in this filter is statistically independent from the next. Because of this reason and the results from literature showing the classification accuracy achieved by filtering email, the false positive rate of the naïve Bayes algorithm was expected to be lower than the other three algorithms. The other filters in this evaluation do not have the assumption of conditional independence between features but take into account the possible correlation of each of the features (where the DMC filter though, does not have features at all).

The first test with the result shown in figure 4.1 was configured using no stemming, and only unigram.

The recall for this setup when using only 500 features is very low but a large increase can be seen for each time . The classification gets better with more features represented, but it is obvious that a continued increase in the number of features would hardly increase the result much more.

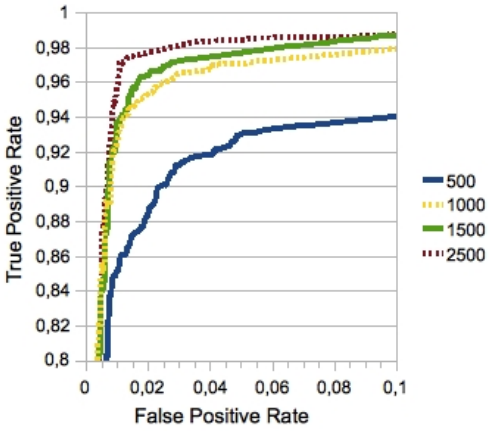

Figure 4.1: ROC curve for Naive Bayes using maxgram 1 and no stemming.

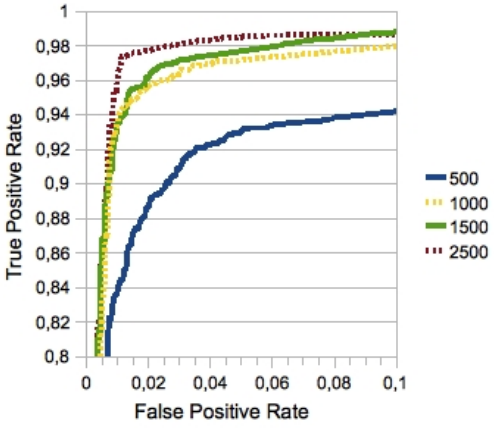

Figure 4.2: ROC curve for Naive Bayes using maxgram 1 and stemming.

The use of stemming in this configuration shows almost no change in the result seen in figure 4.2. Even for a low number of features we can not see any clear gain. It even shows some slight loss in accuracy compared to when stemming is not used. Clearly this setup is not favorable since no clear improvement can be shown. Preprocessing using a n-gram builder is shown in section 4.4.2 to be very slow as well.

The next configuration seen in figure 4.3 used unigrams and bigrams with no stemming.

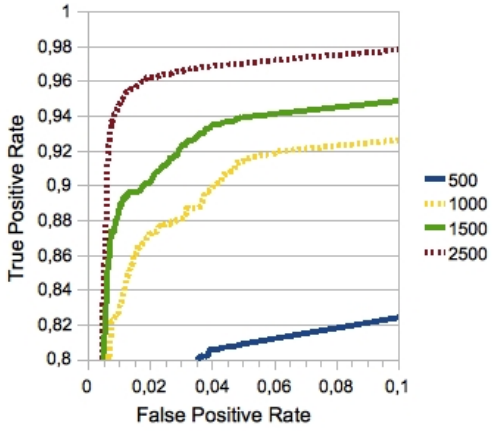

Figure 4.3: ROC curve for Naive Bayes using maxgram 2 and no stemming.

The result was not as expected. With an increased word space because of the use of bigrams it was expected that the precision would increase compared to the earlier unigram tests if enough features were used. As is shown in figure 4.3 the ROC curves are significantly worse in precision than compared to the two earlier tests which only uses unigrams. A likely reason could be that the increased word space because of the bigrams makes it more difficult to match features to a message, thus many filtered messages might have too few features for the filter to classify them correctly by. The two curves with a larger number of features show a big improvement in the precision as the number of features increase. Compared to the two earlier tests, where an increased number of features only gave a minimal improvement this configuration still has room for a more features to further increase the precision and accuracy.

Just as in the earlier test which used unigram and stemming figure 4.4 which uses bigram and stemming shows an early increase in precision and accuracy. Again, the larger word space incurred by using bigrams is both for good and bad. It does give the possibility of getting more information about the type of the message, while it also make it more difficult to match the more specific features in a message given by bigrams compared to finding single words from given from unigrams. Thus more features are generally used to be able to represent more messages, and this I find might be the case here, where more features might possibly improve the result even more. Still I would say the stemming does not do much difference when using large number of features. For large numbers the results are very similar to figure 4.3.

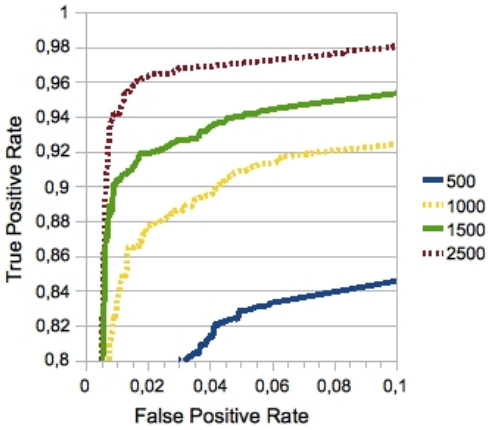

Figure 4.4: ROC curve for Naive Bayes using maxgram 2 and stemming.

The best setup for this classifier was shown to the one using unigrams. It was not clear what gave best results between using or not using stemming since the differences looked marginal. Stemming however would slow the filter down.

4.5.2 Decision Tree

The C4.5 decision tree classifier has shown high classification rates on filtering email data. It is also a classifier that is easy to follow how exactly it is working and why a message is assumed to be one class or another when a message is being classified. This is because of decision trees intuitive constructing. The classifier is easily visualized when the decision tree has been constructed. By being able to examine the visualization it becomes more interpretable, this may be important in helping to understand the classifier, if it will be maintained by domain experts that may not know much about machine learning.

J48, the implementation of the C4.5 decision tree classifier used in WEKA is the implementation tested, the settings used subtree replacement as the pruning option. No subtree raising was used because of the ambiguous results mentioned in the literature regarding this as noted in section 3.2.1.

The first configuration used unigram features and no stemming as seen in the figure below.

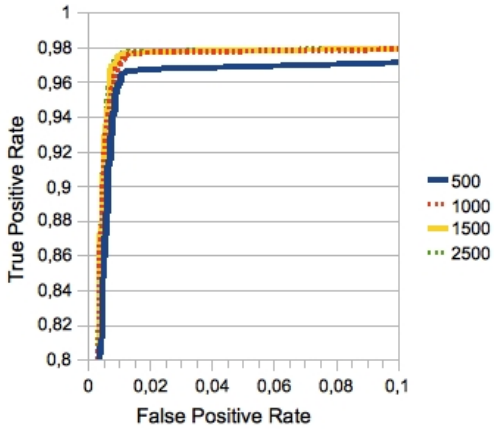

Figure 4.5: ROC curve for J48 using maxgrams 1 and no stemming.

Figure 4.5 shows a continued increase in precision for each increment of the number of features in the 0-1% for the false positive rate. The precision of the C4.5 classifier is better than the the tests for naïve Bayes. It has a higher recall rate while the false positive rate on the false positive-axis is still very much below 1%. The C4.5 classifier outmatches the naïve Bayes classifier in the general scope while it naïve Bayes still show quite close results in the early stage of the curves. Having the classifier taking into account the correlation between the features as well as the possibility to prune the decision tree to remove noise and over-fitting seems to make C4.5 decision tree a better choice for this domain.

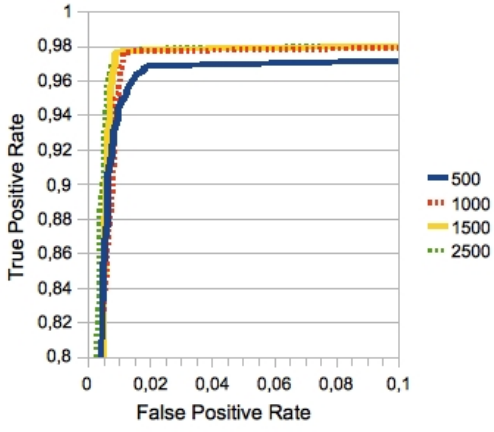

Figure 4.6: ROC curve for J48 using maxgrams 1 and stemming.

Figure 4.6 shows the result of using unigrams and stemming for pre-

processing. What is interesting in these results is that compared to when not using stemming the top accuracy takes longer to reach in the lower n-grams. A reason could be that the dependence between the features becomes distorted by the stemming. The stemming is making the word space smaller, and in the process combines former different features together, thus the former features different dependences gets summed up possibly making erroneous assumptions of dependability. The setup which uses of 2.500 features shows some improvement though. Sadly this setting is way too slow to use if we want it to be able to handle as many messages as required. The slightly better result could be inaccurate because of the confidence interval of the curve.

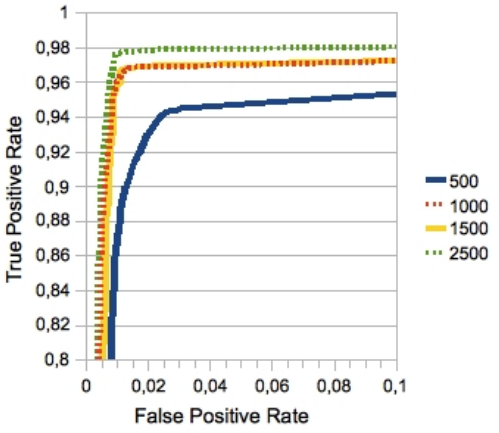

Figure 4.7: ROC curve for J48 using maxgrams 2 and no stemming.

Compared to the results from the naïve Bayes tests, using unigrams together with bigrams for C4.5 decision tree worsen the accuracy as seen in figure 4.7. Even with a higher number of features the results would only marginally increase, possibly the dimensionality in this case is too high for the amount of training data.

The result of using stemming for the C4.5 classifier together with unigrams and bigrams showed only a minimal improvement for the curve in figure 4.8 with the lowest number of features. Apart from that, the curves with the higher number of features incurred a small drop at the most critical section of the curve; that is where the false positive-axis is less than 1%. Even though as a whole, the AUC of the setup using stemming does increase showing a general improvement.

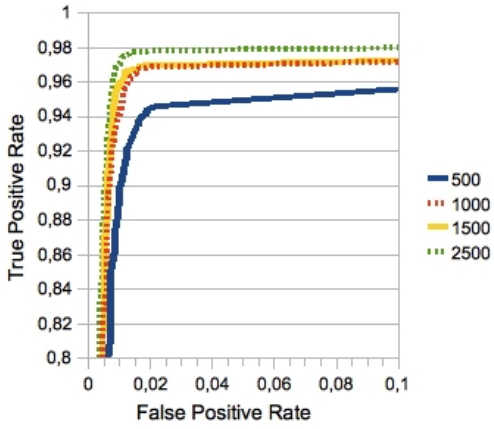

Figure 4.8: ROC curve for J48 using maxgrams 2 and stemming.

It was a surprise that the result from this setup would be worse than the setup which only used unigrams and no stemming. It gives a small improvement for small n-grams but for the largest one. But once again, even if using stemming the dimensionality created by using bigrams together with this classifier could possibly increase the amount of training data needed to find a more correct correlation between the different features used.

The two configurations which showed the best results for C4.5 decision tree were the same ones as for naïve Bayes. Unigram coupled with using or not using stemming. The single best result proved to be unigrams with stemming and a feature number of 2.500. This was deemed to be too slow for the filter though and then the second best results are with a setup not using stemming and a feature number around 1.500. Increasing the number of feature vectors for this setup even more did not give any significant improvement.

4.5.3 Support Vector Machines

The SVM classifier has shown in the experiments done in this study to be one of the fastest, yet also one of the best classifiers in respect to its accuracy. It is also probably the most difficult implementation-wise of the four classifiers and there are several different variations, each of which take a different approach to solving the underlying optimization problem.

The experiments were run with a SVM with linear kernel using the WEKA implementation which in turns uses the SMO algorithm [16].

The first configuration uses unigram features and no stemming.

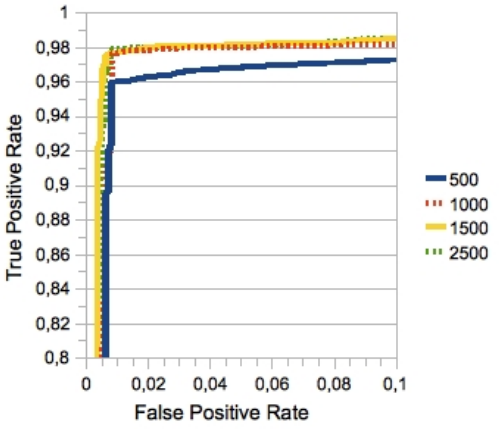

Figure 4.9: ROC curve for SVM using maxgram 1 and no stemming.

Just as the results from the C4.5 tests the results in figure 4.9 are also close to the acceptable range for false positives which should should be near 0.1% while still having a rather high recall rate. The difference between 500 features and 1000 is rather large, but the next increase from 1000 to 1500 is smaller. Oddly enough an increase to 2.500 features gives a worse result. This could happen because the curve is not exact and have some variation which might have given the curve with a bit optimistic results. A continued increase of features make little difference in the accuracy of the classifier.

The results for the following configuration using unigrams and stemming are shown in figure 4.10. As we have seen in previous tests, the stemming *may* help classifiers if you use a smaller amount of features. When working with 500 features, the accuracy does improve, but when the number of features increases, there are diminishing returns from the stemming process. Although marginally, it shows that the recall rate for the classifier with 1500 features increases slower with stemming than without stemming at higher rates of false positives, it is difficult to see any difference at all.

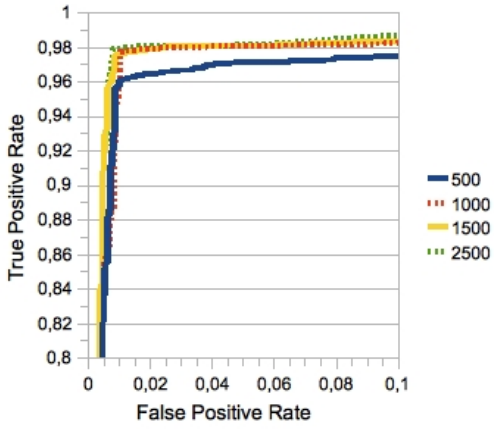

Figure 4.10: ROC curve for SVM using maxgram 1 and stemming.

Figure 4.11 show the results from a configuration using unigrams, bigrams and no stemming

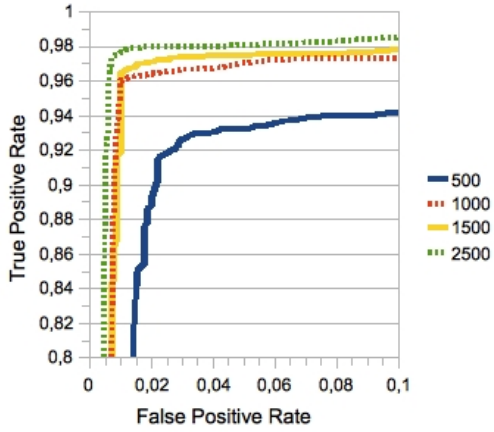

Figure 4.11: ROC curve for SVM using maxgram 2 and no stemming.

Just as seen with the results from C4.5 decision tree, the results with unigrams and bigrams seen in figure 4.11 are worse than only using unigrams. The dimensionality increases as bigrams are used, and comparing the results between the classifier using 1000 features and the one using 1500 features there is a bigger improvement in each step than in figure 4.9 and 4.10. Still though at 2.500 features it has not yet shown any better results than of those setups using unigrams.

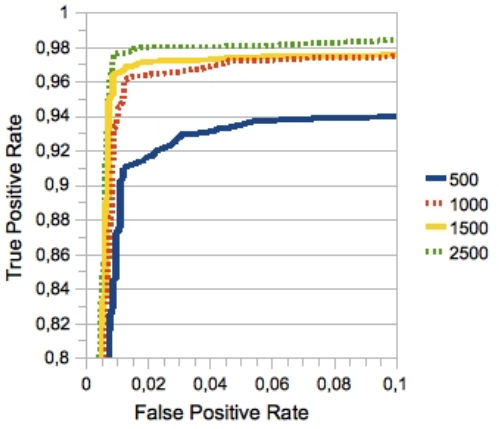

Figure 4.12: ROC curve for SVM using maxgram 2 and stemming.

The setup using stemming with bigrams does give a big improvement in precision for a small number of features as seen in figure 4.12. The results for the setup worsen though compared to not using stemming when using 2.500 features. Looking at the curves given by the unigram setups, there is no real increase between using 1000 or 1500 features. However, it seems again, comparing the two best performing curves in figure 4.12, as if there could still be some increase both in a higher true positive rate, and a lower false positive rate even for low values of the false positive rate. But how much more do the number of features need to be increased for it to show better results than the setups with unigrams.

The best setup for this classifier is when using unigrams and no stemming and either 1.500 or 2.500 features. The number of features did not make a large difference.

4.5.4 Dynamic Markov Coding

There have been reports in the literature [2] that the DMC classifier proved impressive results compared to other established classified, with the highest correct classification rate of all in some of the experiments. This classifier stands out compared to the other ones used in this study, partly because there is no sensible way in which to get an overview of the models used for the classification tasks. Markov chain-models are used for this task as mentioned in section 3.4 and they can be incredibly large depending on the size of the training data. Even if the size of these models would not be very large, they are not very descriptive for the human eye.

Something especially noted about DMC is the fact that while the other classifiers in this study uses a bag-of-words representation, feature selection and in some cases other methods to limit the scope of the training data DMC used no method like that. This was something that would be interesting to

take notice of if it would be possible to see its effects of in the evaluation phase.

As there was no publicly available implementation either in WEKA or otherwise one was created specifically for this experiment. Implementing a DMC classifier is a fairly easy task where the main challenge is the training phase when constructing the Markov chain-models. New nodes are continuously created as the training data is fed. This in turn means that new connections between nodes are created and old connections need to be rerouted. Implementing the main classification part is similar to how classification would work for a decision tree with three differences. The differences are that firstly it is a graph that is travelled instead. Secondly probabilities need to be stored for each time a choice is done to travel through a connection from a node instead of ones at a leaf. And lastly the process is done when the messages has been sequenced as opposed to when a leaf has been reached in a decision tree.

Classification is done by calculating the cross-entropy for each of the models representing either spam or legitimate messages as described in section 3.4.2. The result from the spam model is then divided by the results from the legitimate model, and the logarithmic of the result is calculated. This gives a value ranging from negatives to positives, where zero would be an indecisive value of the classifier. A threshold is set just as the other classifiers, and when the threshold is exceeded for a message it will be classified as spam and if not as a legitimate message. This is the threshold used for construction the ROC-curve.

The curves are labelled by value of the small threshold for expanding the DMC models, followed by the big threshold.

The settings of the DMC classifier compared to the other three, is not as apparent as to what would be a better setting. The value for the thresholds mentioned in section 3.4.1 referred to as the small and the big threshold, governs how DMC should expand its models. Good values for these thresholds are very much a guessing job. Where the other three classifiers usually show some steady improvement while increasing the feature size, you would think that by letting DMC expand its Markov chain models as fast as possible (thus a low value for the big and small threshold) would give the most fitting models, but this is not the case as figure 4.13 shows.

The best results are given by two of the larger thresholds, namely 5,5 and 6,6. It was a surprise that the lower threshold settings would give a worse result than the higher end ones. Having the model expand quicker would yield more information about what makes up, respectively, a spam message and a legitimate message. Although by keeping the threshold values high and thus having a slow expansion, could give a more accurate probability value for each node transition, seeing as the nodes in this case would contain more cases to work on for each transition from a node. Still, even the best of results from the DMC classifier in this case showed worse results than the naïve Bayes classifier, and is far behind both the C4.5 decision tree and

SVM classifier. The poor result might show that it is not suitable to use for messages as short as these are. The experiments for DMC in the literature was done for email messages which are likely most often considerably longer and thus have more to analyze.

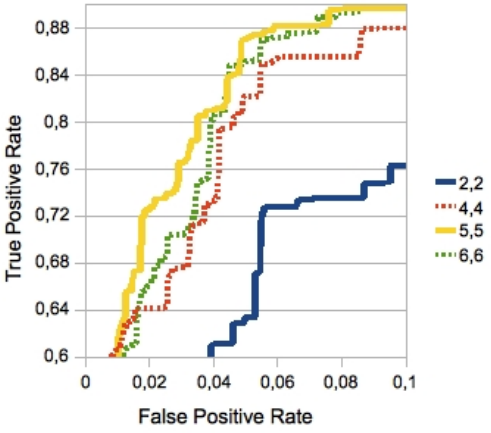

Figure 4.13: ROC curve for DMC, settings for small and big threshold

4.6 Conclusion

It is quite apparent from early tests which of the classifiers that were the real contenders. The DMC classifier had some initial unexpected tests which showed great results. This was only shown though when a small amount of training data was used and is not shown in the experiments. But as the data increased it seemed to have been more of a fluke. The naïve Bayes classifier were achieving quite similar accuracy as the results of C4.5 decision tree and SVM. C4.5 and the SVM classifier however showed the strongest and most stable results, and in the end were the ones with the highest *tpr* for large *fpr* and with an acceptable ratio between true and false positive rate when *fpr* was below one percent.

The result of C4.5 and SVM are very similar when comparing the best curves in the figure 4.14. Both shows a large *tpr* for *fpr* at 1% and above. The setup decided to show the best accuracy for the ROC curves is the unigram without stemming-setup.

In the tables 4.10, 4.11 and 4.12 we can see a more detailed view on the results seen in the figure. The tables shows the 95% confidence interval for the C4.5 decision tree and SVM classifier at relevant *fpr* values and with a varying number of features.

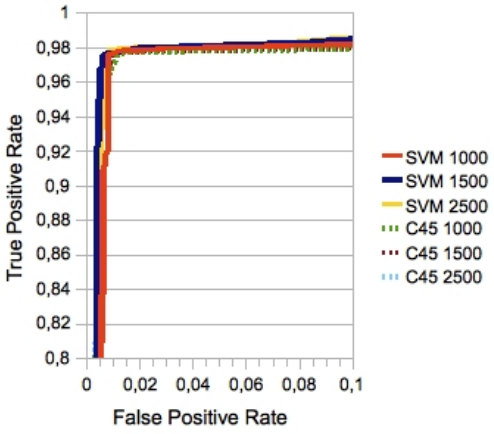

Figure 4.14: Results from the two contenders. SVM and C4.5 decision tree using unigrams and no stemming with 1.000, 1.500 and 2.500 features.

The results from the tables 4.10, 4.11 and 4.12 shows much broader confidence intervals for small *fpr* compared to larger ones. In most of the tables C4.5 has a small lead over the SVM classifier. They have similar confidence intervals in each test. SVM shows worse accuracy when the number of features are increased to 2.500. This is possibly because of overfitting and might be improved by tuning the regularization parameter mentioned in section 3.3.1.

The experiments showed that the best setup was when using 1.500 features, unigram and no stemming. It proved to give a good precision and accuracy as well as speed. 2500 features might be wished for to get a slight increase in accuracy, but it could also make the filter slower. The choice of not using bigrams and stemming also gives a decrease in classification time. This was the reasons for the preferred setup.

Performance wise the C4.5 classifier is faster than the SVM classifier to classify an incoming message. Both however are well above the requirements at these data sizes. They are similar in speed and memory consumption except for C4.5 being slower when building. This is a process that should not need be done very often though.

Comparing accuracy we can see that the value for *tpr* varies quite a bit for low values of *fpr* but they are similar in their confidence levels.

Looking at it from an implementation viewpoint, C4.5 seems more straightforward. Decision tree construction in general is a common task while the implementation of a quadratic programming problem seems to me less intuitive and a trained SVM-classifier may be more difficult to interpret for non-experts.

For this reasons the classifier that was chosen to be implemented was the C4.5 classifier.

FPR:	0.1%	0.4%	1.0%
C4.5	0.228-0.572	0.802-0.928	0.952-0.986
SVM	0.012-0.224	0.489-0.921	0.973-0.979

Table 4.10: This table shows the confidence interval of the *tpr* for given *fpr* positions. The SVM as well as the C4.5 decision tree classifier uses unigram and no stemming with 1.000 features.

FPR:	0.1%	0.4%	1.0%
C4.5	0.216-0.580	0.809-0.943	0.973-0.979
SVM	0.123-0.467	0.863-0.981	0.972-0.980

Table 4.11: This table shows the confidence interval of the *tpr* for given *fpr* positions. The SVM as well as the C4.5 decision tree classifier uses unigram and no stemming with 1.500 features.

FPR:	0.1%	0.4%	1.0%
C4.5	0.239-0.523	0.800-0.940	0.969-0.981
SVM	0.091-0.431	0.695-0.945	0.974-0.982

Table 4.12: This table shows the confidence interval of the *tpr* for given *fpr* positions. The SVM as well as the C4.5 decision tree classifier uses unigram and no stemming with 2.500 features.

Chapter 5

Implementation

In this chapter an overview and some of the more interesting parts in the implementation and the thoughts behind them will be represented as well as an overview. After having evaluated the possible classifiers, it was decided to implement a version of the C4.5 classifier itself. To achieve this it was important to understand well not only how the classifier was designed, but also the pre-processing part. For the C4.5 classifier there were already two existing implementations, in addition to the already studied literature, to dive in and examine. Both WEKA's J48 [11] implementation in Java and the open source implementation of C4.5 [18] in C were very helpful in trying to finish the implementation.

The filter is implemented at the short message service center stage in the network which is mentioned in section 1.2.2 about the mobile phone network.

In figure 5.1 there are several filters for incoming messages, so the decision tree filter is only one of several evaluating messages. If any of the filters would classify this message as inappropriate, it would be discarded. Otherwise the message will be stored in the short message service center until it is sent forward to the recipient.

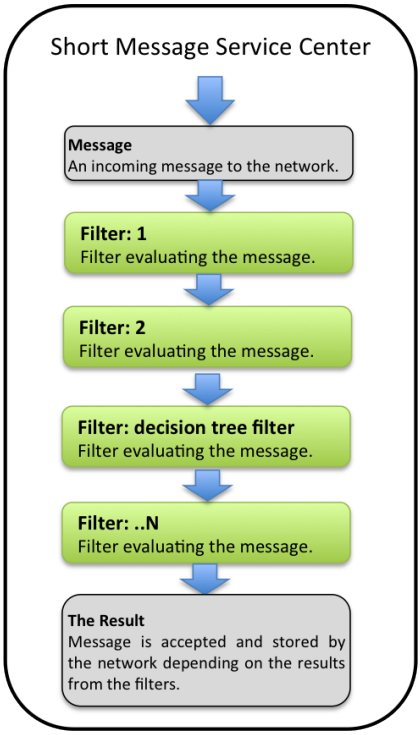

Figure 5.1: An overview of where the filter is placed in the network.

5.1 Programming Language

The language used for implementation was C++, and the compiler was G++ [9]. The version of the compiler used was 4.6.2 to have the necessary support for the new string type u16string. This was necessary in order to map the filter implementations message data type to the server's that it was implemented on.

5.2 Overview

There are two major steps in this filter shown in figure 5.2. There is the pre-processing and the classifier step. Except for this there are also two modes of the filter, the training mode and the filtering mode.

In the training mode the first thing that is done is to load the training data, which here is the necessary classified messages. The pre-processing step consists of creating a feature vector template and then building the feature vectors for the training data. The first part of this consists of firstly tokenizing the incoming messages, and then do a feature selection over all

the tokens using information gain which is discussed in the *Feature Selection* section 2.1.5. The top n features with the highest information gain will be chosen as valid features in this filter, where n is the number of features asked for during the setup of the filter training. The feature selection part is the most time consuming part of training the filter.

The second part in pre-processing, the message representation part to construct feature vectors from the training messages. It begins by trying to match the tokens from a message to a valid feature in the feature template. When a token from a message matches a valid feature, the feature count will be incremented by one, starting from zero if there are no tokens matching that feature. When all the tokens have been checked a feature vector for the current message has been constructed and it starts over with the next message in the training data until all messages have been a representation.

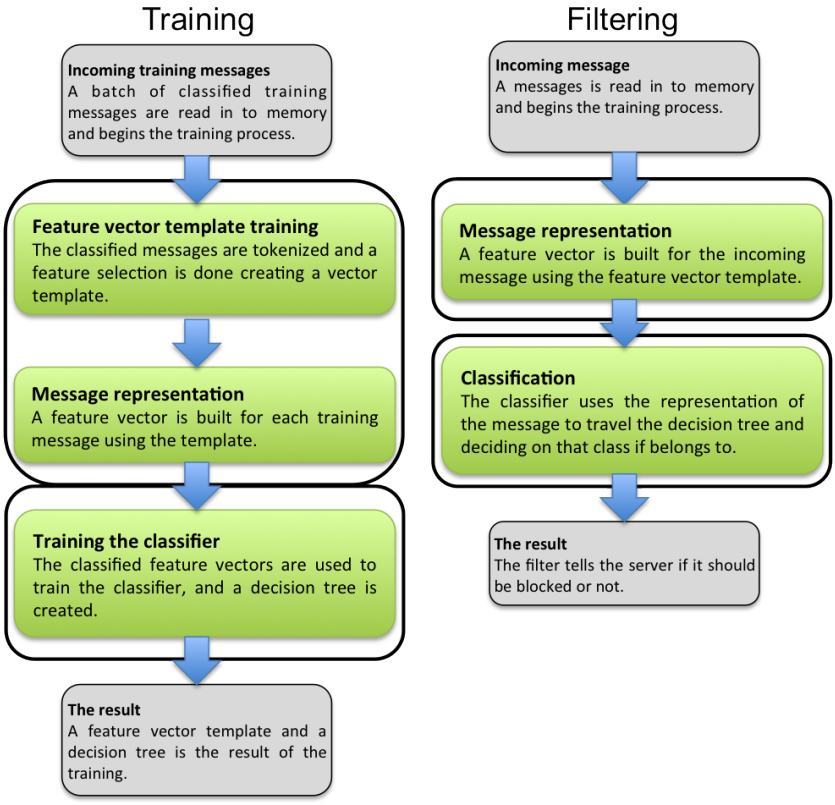

Figure 5.2: An overview of the filter, showing the parts for classification and learning. This includes tokenization, feature selection as well as decision tree construction and classification tasks.

The last part in training is to receive the classified feature vectors, which has just been constructed from the training data, and send them to the

training classifier part. Here the vectors are used to construct and optionally prune a decision tree following the steps discussed in section 3.2. When done a feature vector template and a classifier has been constructed, and the filter is ready to be used.

The filtering mode only has one incoming and unclassified message at a time. It is unclassified since of course we can not know beforehand what type a message is which is not a part of the training data. This phase contains a pre-processing and a classification step.

The pre-processing stage contains a message representation step, which is carried out exactly as mentioned in the pre-processing stage.

When a representation of the message is finished, it is sent to the classifier which is the decision tree, to be evaluated. When the evaluation is done the filter will ask the server to either keep forward the message or to discard it.

5.3 Pre-processing

There are two major parts of the pre-processing, it is the *feature vector template training* which is only found in the training mode, and there is the *message representation* parts which is found in both stages. An overview for each of these parts are shown in the figures 5.3 and 5.4.

The two parts seem similar at first since both tokenize their messages. But to construct a feature vector template it is assumed that a batch of incoming classified messages are fed at once, to be able to complete the feature selection step. With only a single message or a batch of messages where each one belongs to the same class would of course make feature selection impossible. This step would in such a case result in no grading for different features. Look at the *Feature Selection* section of chapter 1 for more on this.

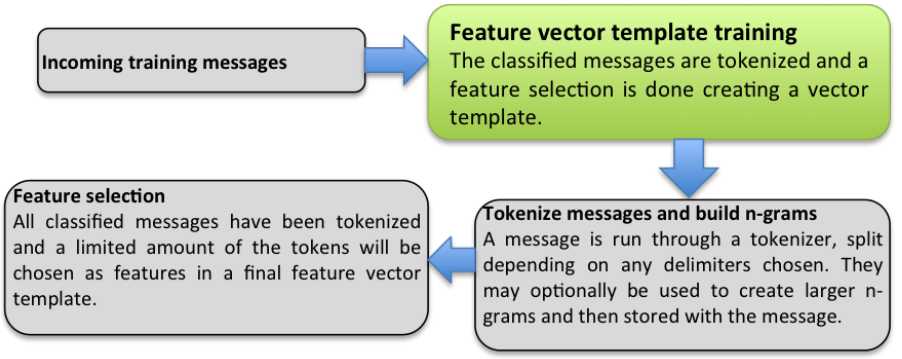

Figure 5.3: Steps on how to build a feature vector template in the implementation. This includes tokenization, n-gram construction and also feature selection.

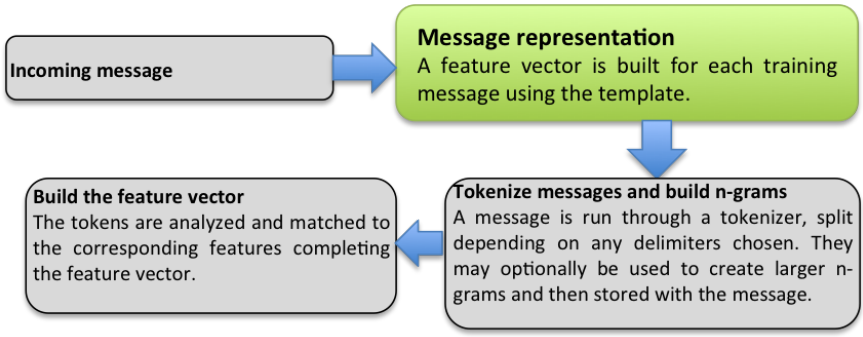

Figure 5.4: Steps on how to build feature vector in the implementation. This includes tokenization, n-gram construction and creating a vector representation

The message representation step does not have the this assumption. Messages are processed and sent forward one at a time for this step and it has no need to know if a message is classified or not. When building a feature vector though, a template for feature vectors must already exist this step relies on the *feature vector template training* to have completed. If this is not the case the feature vector builder can not assign any features since there have been no valid features to use yet.

A more detailed discussion for the tokenization, the feature selection and the feature vector builder are in the following subsections.

5.3.1 Tokenization

The sections 2.1.2 about tokenization and 2.1.2 is relevant for this area of discussion. The tokenizer is implemented to read the string consisting of the message to start from the first and end at the last character. There are two positions stored at all time, a start position and the end position. The algorithm 1 first search for the position of the first character which is not a delimiter, starting the search at the end position. Next it start search for the first character which is a delimiter starting from the start position and sets the end position to this value. A token is saved consisting of the characters between the start position to the current one. The algorithm then continue repeating these steps until the start position is the same as the end of the message.

This was a very straight-forward implementation and not much thought needed. Some things to mention are that the tokenizer only handles delimiters of length one. That means it does not handle a delimiter value consisting of several character, but it does handle several delimiters in a row. Thus it cannot handle a specific pattern delimiter but can only analyze on character each step. The tokens are also stored in the order they were extracted, this is critical for the n-gram builder to work.

```
Data: begin,end=0
Data: message
Result: tokens
while begin is not end of message do
    begin = nonDelimiterCharacterPosition(message,end);
    end = delimiterCharacterPosition(message,start);
    tokens += startToEnd(start, end, message);
end
```

Algorithm 1: Tokenization algorithm

In this implementation the n-gram constructor is a part of the tokenizer, the algorithm can be seen below as algorithm 2. N-grams are optionally constructed after the message has been tokenized and does so by reading the tokens from first to last. It sequentially adds every token between the i-th token to the i-th+n token together to construct a new n-gram which is stored with the message object. Here i is the current token number in the loop and n is the wished size of the expected n-gram. This loop is repeated starting from the min The size of the n-grams are an option for the training of the filter.

```
Data: originalTokens
Data: n-gramTokens
Result: n = mingram
while n is smaller or equal to maxgram do
    while i+n is smaller than or equal to #originalTokens do
        n-gramTokens.add(originalTokens(i) to originalTokens(i+n);
    end
end
originalTokens = n-gramTokens;
```

Algorithm 2: N-gram algorithm

5.3.2 Feature Selection and Message Representation

The first problem encountered, which needed some assistance, was already before the feature selection stage of this implementation. The problem was how to represent messages as feature vectors before any feature selection was done. To have each feature vector object contain a huge vector with indexes to every possible token proved to consume a large amount of memory and was not very efficient.

Knowing that experiments had already been carried out in WEKA building feature vectors with the training data that was going to be used with this implementation, we investigated how this problem had been solved there. The solution was simply to have a unique identifier for each word encountered, where the identifier was the word. This was then used to map each word to a counter, which increase by one each time another identical word was found in the same message. Thus there was a map with word to counter

structure.

It was then expanded on, so that when a feature vector template was constructed, a word in the template was simply mapped to a index value from 0...n where n is the total number of feature. So then a feature template when constructing a feature vector added features to the vectors by mapping a word's index value to a counter instead of a word to a counter. The feature vector then got its own mapping structure with index number to word count instead. The idea was that it would be simpler to handle the access of features from a feature vector using index numbers from 0...n to catch the i-th feature than to iterate through a map structure. This made the implementation of the classifier feel more intuitive when communicating to a feature vector.

The other idea behind it was that when a feature was being processed by the classifier, it would be faster to do number comparisons than string comparisons when accessing a certain feature.

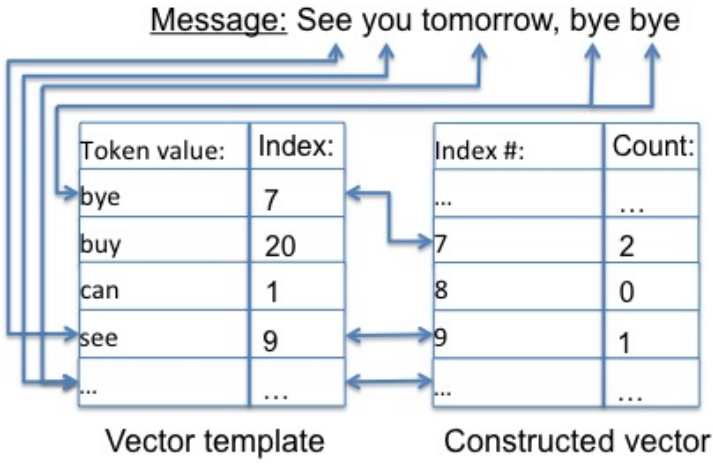

Figure 5.5: The structure for the feature vector template word to index, and the feature vector index to word count structure.

From the figure the implementation structure can be seen. When constructing a feature vector, the words are mapped to preset index numbers which are created during the template training. When having the index for a certain word found in an incoming message, the count can be added to the feature vector for the corresponding feature.

Both the feature selection and the message representation step assume that an incoming message is already tokenized. The pseudo-code seen in algorithm 3 shows how a feature vector template is constructed using a feature selection method After the training messages have temporary feature vectors, each token would be given a distribution count, saying how many messages with a certain token that were spam messages, and how many messages that were legitimate messages. This is illustrated in algorithm 4.

```
Data: chooseNFeatures
Data: allMessages
Data: currentMessage
while currentMessage is not end of allMessages do
    while currentMessage has more tokens do
        currentToken = currentMessage.getNextToken();
        if globalTokenIndex does not contain currentToken then
            globalTokenIndex += currentToken;
        end
        currentMessage.addToken(globalTokenIndex.getIndex(currentToken));

        currentMessage = allMessages.next();
    end
end
```

Algorithm 3: Feature vector template construction 1

```
Data: currentMessage
foreach currentMessage in allMessages do
    foreach token in currentMessage do
        if currentMessage = "spam" then
            tokenDistributions[token][SPAM]++;
            totalCountForType[SPAM]++;
        end
        else
            tokenDistributions[token][LEGITIMATE]++;
            totalCountForType[LEGITIMATE];
        end
    end
end
```

Algorithm 4: Feature vector template construction 2

After this the information gain score for each token was easily found by using the distributions given, and each token could be ranked for the feature selection as in described in the *Feature Selection* section of chapter 1. This is illustrated in algorithm 5.

When each feature have been given a score the global index is sorted in descending order from highest to lowest score. The features at position $chooseNFeatures$ and below are all removed. Next each feature is given a index number which will be the number representing the feature in a message's own feature number. That means that each feature for a message's feature vector will only be represented by a number from $1...chooseNFeatures$. When a classifier later asks for a certain one it simply just asks for a number.

```
Data: chooseNFeatures
Data: allMessages
Data: currentMessage
...
Data: mapTokenToScore
forall the token in globalTokenIndex do
    mapTokenToScore += informationGain(token);
end
sortDescending(mapTokenToScore);
for index=chooseNFeatures to #mapTokenToScore do
    remove(mapTokenToScore, index);
end
clear(globalTokenIndex);
Data: indexNumber = 0
foreach token in mapTokenToScore do
    globalTokenIndex.add(token, indexNumber);
    indexNumber++;
end
```

Algorithm 5: Feature vector template construction 3

5.4 Classifier

As can be seen in the overview, the classifier has only two major parts, either to train the classifier in the training phase or to apply it in classification problems in the filtering phase.

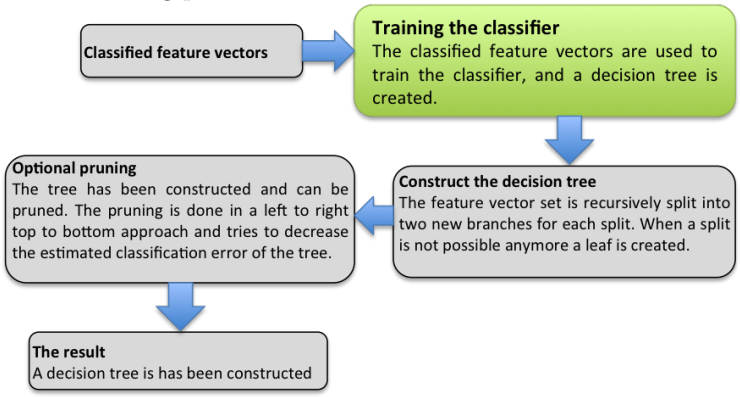

Figure 5.6: The flowchart of the major steps of training a decision tree. This includes node splitting, leaf creation and pruning.

When training the classifier there needs to be more than one classified message with a presentation that the classifier can understand. In this implementation the presentation is feature vectors. The training of the classifier will not be possible with a single message or a single-class collection

of training messages for the same reason as mention in the previous section *Feature Selection*. If that is fulfilled there are two thing to do to train the classifier. Firstly it is constructing the decision tree. This is done by testing to split the current training data on different features and different feature values, and grading each split using information gain and information gain ratio. The feature together with the feature value achieving the highest information gain ratio is chosen and the process is continued until no more splits are possible, then a leaf will be created. This is just the same as in the *Decision Tree Learning* section of chapter 3 on how splitting is done.

This proved to show some obstacles, especially how to split and move messages up to different branches of the tree. Since the implementation for building the tree was recursively done for simplicity of implementation, if all necessary messages were to be copied into the next recursion memory could increase much and would be unnecessarily slow. This was solved though and the results can be seen in subsection 5.4.1.

The next is the optional pruning in the implementation. The pruning is done in a left to right top to bottom approach and tries to decrease the size of the tree and the estimated classification error of the tree. When these steps are done a ready decision tree is the result.

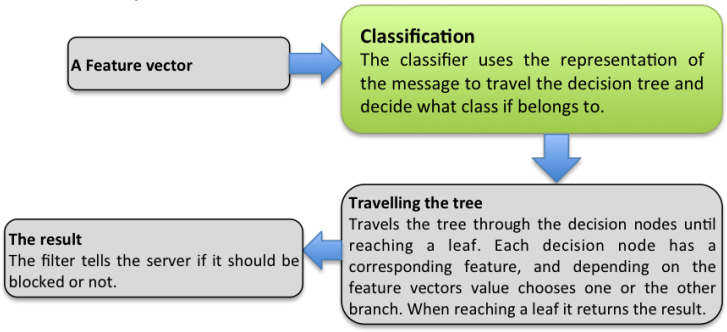

Figure 5.7: The flowchart for the steps of the classifier. This includes node traveling and leaf results.

When classification is done the only thing expected is that the message has a correct representation for the classifier to read. The message will travel, starting from the root node until it reaches a leaf. It navigates through the different branches as described in the section 3.2. When a node is reached in this implementation, a probability is given for the first class type in the training data no matter if it is the majority class type in this leaf or not. During the experiment spam messages were always classified as the first class type, or as class type '0'. So when the probability which is a value between 0...1 is given for for a message, it tells how likely it is to be of the first class type. This is then used by a threshold value which can be changed. If the probability is more than or equal to the threshold, the filter will tell the server to block the message, otherwise it will report nothing.

The threshold makes it possible to vary how strict the filtering should be. This is a highly important to control the general rate of false positives and true positives of the classification task. If the probability given by a leaf would be equal to or larger than the set threshold the result would be to tell the server to discard the message, otherwise it is just forwarded.

5.4.1 Training the Classifier

The tree structure is implemented to be built of a number of nodes. Each node is an independent object which can either represent a decision node or a leaf. Each decision node contains a flag telling us it is a decision node, the feature vector it splits on, the value which it is split on and two references for its child nodes. Thus each split is a binary split just as the one mentioned in section 3.2.1 has been. If the node would represent a leaf instead the same flag as mentioned, will be set to tell us that it is a node. It will contains a probability telling us how likely it is that a message reaching this node belong to the first class type.

```
Input: currentNode, trainingData
buildNode(currentNode, trainingData)  Data: splitInformation
splitInformation = getSplitInformation(trainingData);
if splitInformation.action == LEAF then
    currentNode.type = LEAF;
    currentNode.probability = calculateProbability(trainingData);
    return;
end
Data: rightBranchTrainingData
;
forall the message in trainingData do
    if message.featureValue(splitInformation.feature) larger than
    splitInformation.featureThreshold then
        rightBranchTrainingData = message;
        trainingData.remove(rightBranchTrainingData);
    end
end
buildNode(currentNode.leftBranchNode, trainingData);
buildNode(currentNode.rightBranchNode, rightBranchTrainingData);
```

Algorithm 6: Decision Tree construction algorithm

Algorithm 6 shows how the tree is being constructed recursively node by node going through each left branch before the right branch. The decision for if to split a node or not is decided by calling the *getSplitInformation* method. The method analyses if a split is possible by seeing if there is enough training data available. To allow a split, a minimum amount of training messages has to be exceeded. If this is the case it calculates the information gain and information gain ratio for every feature by being fed

the training data in the current node. If it finds that a split is possibly it returns which features and which feature values the node should split messages on. The feature with the highest information gain ratio is then chosen. If several splits have the same ratio the first one is chosen. Each possible split must also have an information gain higher than the average information gain of all the possible evaluated splits for it to be chosen. If doing a split is not possible, then it will tell the *buildNode* algorithm to make a leaf out of the node.

When the current node becomes a decision node the training data are split into two groups, the messages with the current feature having a value over the split threshold is moved to the right branch, while the others are moved into the left. And so the recursion continues every branch is stopped at a leaf.

```
Input: currentNode
Data: errorlargestBranch = 0
Data: errorNodeAsLeaf = 0
Data: errorSubTree = 0
if currentNode.type != LEAF then
    prune(currentNode.leftBranchNode);
    prune(currentNode.rightBranchNode);
    if usingSubtreeRaising then
        errorlargestBranch = errorLargestBranch(curretNode);
    end
    errorNodeAsLeaf = errorForNode(currentNode);
    errorSubTree = errorForTree(currentNode);
    if errorNodeAsLeaf less than or equal to errorSubTree AND (not
    usingSubTreeRaising OR errorNodeAsLeaf less than or equal to
    errorlargestBranch) then
        currentNode.type = LEAF;
    end
    else if subTreeRaising AND errorlargestBranch less than or equal
    to errorSubTree then
        Data: minorBranchTrainingData
        minorBranchTrainingData =
        currentNode.getMinorBranchData();
        redistributeTrainingDataToSubtree(currentNode.getLargestBranch,
        minorBranchTrainingData);
        currentNode = currentNode.getMajorBranchNode();
    end
    return;
end
```

Algorithm 7: Decision Tree pruning algorithm

When the tree has been created the optional pruning process is run. The pruning process has two major things to do. The first is to find out the theoretical errors for the current node, the subtree and the largest branch.

The second is to make a decision if pruning should be done at this node or not.

The pruning is done as a recursive function going from top to bottom of the tree and from left to right. At leafs there is obviously no pruning possible and as such at that step it will back the recursion.

The first thing seen in algorithm 7 is that if the current node is a leaf the method will instantly return. Otherwise the method prune is recursively called on for the branches of the node, to climb as high up into the tree as possible. Then the pruning begin by calculating the three necessary errors for deciding if the pruning should be stopped, if there should be a subtree replacement or lastly if subtree raising should be done.

Subtree raising is only done in pruning if specifically enabled. The subtree pruning chooses the branch with the least amount of training data leading to it, and redistributes the data to the larger branch. Thus there will then be new probabilities in the leafs because of the new data reaching there. This subtree is otherwise not modified. The larger subtree root then replaces the current node and the pruning for this step is completed.

When this process has been finalized, the result is a built pruned decision tree ready for classification.

5.4.2 Classification

The classification step shown in algorithm 8 is fairly simple. It is of course assumed that a classifier is available. It is also assumed that the message which is being classified has a proper representation. That would be a feature vector which was built by the same feature vector template as the training data was when the classifier was constructed.

It is not much to say except that the classification algorithm 8 is implemented as a non-recursive function unlike the algorithms for constructing and pruning the decision tree. This choice is just made out of convenience. The implementation process is easy enough for any of the choices but this way was the first that came to mind. It might be that a recursive function would have been slightly slower because of the function call overheads that might incur. Either way the difference would hardly be noticeable.

A loop is run as long as *currentNode* is not a leaf. The current decision node analyze the message's feature vector on the relevant feature the node branches out on. If the feature's value from that feature vector is less than or equal to the decision nodes threshold value, the currentNode will become the left child node, otherwise it will become the right one. The loop is then repeated until *currentNode* is a leaf.

The leaf returns the probability value that the message belongs to the first message class of the two possible classes. This value is compared to an agree threshold, and if the value is below the threshold, the message is classified as belonging to the second class. Otherwise the message is classified to belong to the first class. When this step is finished the filter has

done its job and the process is repeated again for any new incoming message.

```
classify(currentNode) Data: currentNode = rootNode
while currentNode.type is not LEAF do
    Data: feature = currentNode.splitFeature
    Data: threshold = currentNode.splitThreshold
    if message.feature(feature) is less than or equal to threshold then
        currentNode = currentNode.LeftBranchNode();
    end
    else
        currentNode = currentNode.RightBranchNode();
    end
end
return currentNode.probability();
```

Algorithm 8: Classification algorithm

5.5 Results

The best setup for the decision tree classifier in this environment was found by tests to be 1500 features, unigram and no stemming. It showed a fast processing speed coupled with an acceptable accuracy. Generally when increasing the number of features the accuracy and precision is expected to increase up to a certain level, so it was expected that the setups having one of the largest amount of features would perform best. At the same time an increased number of features will also make the filter process each message slower.

However, stemming did not give much better accuracy for the decision tree classifier and showed some varying results. It was assumed the possibility that erroneous assumptions of dependability between words that have now been stemmed, being given a dependence which would not have existed would the word have kept its original form and not only its root form. Though SVM showed similar results without having this possible erroneous dependability so the assumption could be incorrect. Stemming considerably lowered the number of messages that could be classified for a given time. Though the implementation of stemming might be possible to optimize it will still be one of the most time-consuming processes in these configurations.

Using bigrams together with unigrams showed to worsen the results of the C4.5 classifier, one reason for this could be that the amount of training data was too low or the number of features were too low to properly represent all possible messages in this increased word space.

5.5.1 Accuracy

Tests were done on the implementation to figure out what the performance is. A setup of 1500 features using unigram and without stemming had proven to give the best result and as such it would be interesting to test on the implementation.

As can be seen in figure 5.8, the precision and accuracy was as I had wished for. Having such a low false positive- while retaining a high true positive rates means that it is getting close to a filter which could be accepted in practical use for this domain. The desire was that the false positive rate would be much less than 1%, as 1 percentage of all sent SMS messages is far from acceptable.

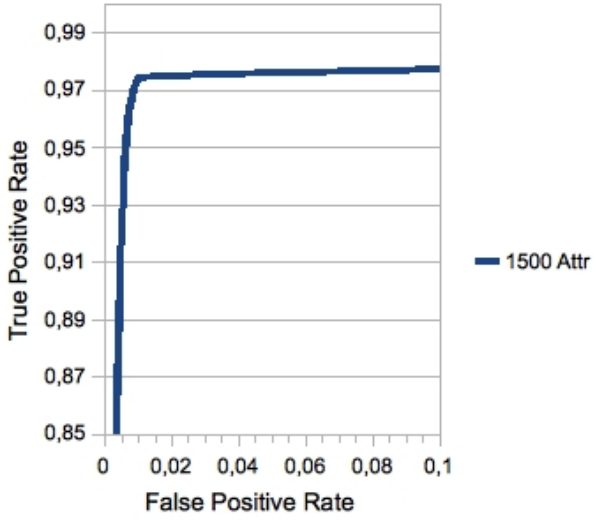

Figure 5.8: ROC curve for the implemented classifier.

The filter at a certain threshold showed a 0.1% false positive rate and catch rate of spam at around 52% with a confidence interval at about 20%. It is doubtful though that the catch rate is high enough to justify its use in spam filtering in comparison to the already existing filter. The confidence interval also shows that the average catch rate is not very exact. At a false positive rate of 0.4% about 85% of all spam was caught with a confidence interval of 9%. Lastly at 1% for the false positive rate 97% of all spam is caught with only a confidence interval of 0.7%. To reach up to a catch rate of over 90% the rate of legitimate messages lost becomes unacceptable. What has been wished for is a false positive rate of around 0.1% while stopping around 90% of all spam.

The decision tree classifier with the mentioned settings comes very close to achieving an accuracy and precision which would be fully acceptable, but there is a wish for achieving a bit lower false positive rate while keeping the

high recall for being completely acceptable for this use.

5.5.2 Speed

The speed of building the classifier was a bit faster than the WEKA implementation. It took about 15 minutes to build and prune the decision tree with this implementation using the same training data and selecting 1500 attributes. The build time of course varies slightly on this for different data but it should be somewhat similar with the same a number of messages used to train with.

The speed of classifying messages lies at about 100.000 messages per second, excluding the time tokenization and feature vector build-up which by itself processed about 10.000 messages per second. This is quite a bit faster compared to the speed given by the test data when using the WEKA implementation. This implementation is quicker but most importantly it is fast enough to be used as a real time filter with this configuration.

Chapter 6

Conclusions

The goal of this work is to evaluate a number of machine learning algorithms which could function as SMS-filters. The filters should be evaluated on their memory usage, time for training, classifying messages as well as accuracy. The filter judged to show the best results is to be implemented and tested on real data to compare its performance.

Based on the initial evaluation the decision tree filter was considered the most promising of the four filters evaluated. However, the full evaluation of it showed that it may not have been as high as initially hoped. Depending on where you set the limit on the accuracy for an acceptable loss of messages contra the gain of ridding the network of unsolicited advertising.

The memory usage for the decision tree filter was never an issue since it was well below the agreed on maximum. The decision tree filter just as the other ones need to be trained on real and current message data to function properly. The training time did differ largely between different filters however the filters in this environment are not expected to be trained very often so even a filter with a rather long training time like the decision tree filter is acceptable.

The evaluation could possibly have been improved however by examining which specific messages that were caught on the wrong side of the filter. This could be investigated by running a session and saving incorrectly classified messages at some given thresholds. The messages could then have been given more weight in the current training data and then once again train a new filter. This would hopefully teach the filter to classify these cases more correctly while classifying the other messages the same as before. It is not clear at this moment if it would have increased the accuracy significantly.

But while the aspect of using the C4.5 decision tree filter against SMS-spam do show results to an extent, it has to be compared to the precision and accuracy of other types of filters already in use by Fortytwo Telecom. Even if the filter is not used for spam filtering it could be used for other tasks instead. Examples could be to prioritize certain traffic when the network is

over-loaded.

However there is another aspect with these kinds of filters. If a filter like this is used commercially in the telecom network, there is the question of how often, and how you would go about updating it. Spam change in their presentation over time to try and avoid filters like this so that is why these filters needs to be updated from time to time. This filter will need to be maintained and possibly tested on new traffic once in a while, as well as continuously saving a portion of the data flowing through the network to use it in future training.

This could possibly be a big task, since the data will need to be classified before using it to train the filter. The workload on this task would likely completely depend on how often spam messages transmute to effectively avoid filters. Of course tools could help to speed up the process of classifying the new training data. A tool could take help from an already existing trained classifier to partly assist in classifying new training messages. Since many spam messages have nearly identical content, the most similar ones could be grouped and classified manually by only a few clicks of the user.

Of course machine learning algorithms such as this one have the advantage that it can be very adaptable by simply using different training data. This means that it could potentially give a quick transition from this current domain for spam to another one such as prioritizing certain data traffic if wished for.

6.1 Future Improvements

Since the precision of the filter turned out to be a bit less than what could be asked for, it would be interesting to see if there could be any improvements done to change this. Two methods that I have read about that claims to do just this, to substantially improve predictive accuracy, are bagging and boosting [17]. Both of these methods work by training several models for a classifier and combining them together to create a stronger one. It would be interesting to see how the results would differ after using any of these methods.

Only tests on linear SVM classifier was made and it would be interesting to see if there would be any improvement with a different kernel than the linear one. Since the results from the SVM classifier were so similar to the C4.5 decision tree results in respect to both accuracy and speed, it is still definitely something to consider.

It would also be interesting to find out how the setups using bigrams would improve with a further increased number of features, and thus also an considerably increased amount of training data.

Bibliography

[1] Enrico Blanzieri and Anton Bryl. "A survey of learning-based techniques of email spam filtering". In: *Artif. Intell. Rev.* 29.1 (Mar. 2008), pp. 63–92. ISSN: 0269-2821.

[2] Andrej Bratko et al. "Spam Filtering Using Statistical Data Compression Models". In: *J. Mach. Learn. Res.* 7 (Dec. 2006), pp. 2673–2698.

[3] Christopher J. C. Burges. "A Tutorial on Support Vector Machines for Pattern Recognition". In: *Data Min. Knowl. Discov.* 2.2 (June 1998), pp. 121–167.

[4] Gordon V. Cormack. "Email Spam Filtering: A Systematic Review". In: *Found. Trends Inf. Retr.* 1.4 (Apr. 2008), pp. 335–455.

[5] Adam Krzyzak Jian-xiong Dong and Ching Y. Suen. "A Practical SMO-Algorithm". In: *The 16th International Conference on Pattern Recognition (ICPR2002)*. Centre Of Pattern Recognition and Machine Intelligence Concordia University, Aug. 2002.

[6] Tom Fawcett. "An introduction to ROC analysis". In: *Pattern Recogn. Lett.* 27.8 (June 2006), pp. 861–874.

[7] Tristan Fletcher. "Support Vector Machines Explained". In: University College London, Dec. 2008.

[8] *Fortytwo Telecom*. 2012. URL: `http://www.fortytwotele.com`.

[9] *GCC, the GNU Compiler Collection*. 2012. URL: `http://gcc.gnu.org/`.

[10] Thiago S. Guzella and Walmir M. Caminhas. "Review: A review of machine learning approaches to Spam filtering". In: *Expert Syst. Appl.* 36.7 (Sept. 2009), pp. 10206–10222.

[11] Mark Hall et al. *The WEKA Data Mining Software: An Update*. Vol. 11. 1. SIGKDD Explorations, 2009.

[12] Brian Hayes. "How many ways can you spell Viagra?" In: *American Scientist* 95 (2007).

[13] F. Heylighen and C. Joslyn. *Entropy and Information*. Principa Cybernetica Web, 2001.

[14] Jos M G Hidalgo and et al Cajigas Guillermo. "Content based SMS spam filtering". In: (2006).

[15] Andrew McCallum and Kamal Nigam. *A comparison of event models for Naive Bayes text classification*. AAAI Press, 1998, pp. 41–48.

[16] J. Platt. "Fast Training of Support Vector Machines using Sequential Minimal Optimization". In: *Advances in Kernel Methods - Support Vector Learning*. Ed. by B. Schoelkopf, C. Burges, and A. Smola. MIT Press, 1998.

[17] J. R. Quinlan. "Bagging, Boosting, and C4.5". In: *In Proceedings of the Thirteenth National Conference on Artificial Intelligence*. AAAI Press, 1996, pp. 725–730.

[18] J. Ross Quinlan. *C4.5: programs for machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[19] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

Avdelning, Institution
Division, Department

IDA,
Dept. of Computer and Information Science
581 83 Linköping

Datum
Date

2013-05-16

Språk
Language
☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category
☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
-

ISRN
LiU-Tek-Lic–13:021-SE

Serietitel och serienummer **ISSN**
Title of series, numbering -

Linköping Studies in Science and Technology

Thesis No. 021-SE

URL för elektronisk version

http://XXX

Titel
Title

Spam filter for SMS-traffic

Författare
Author

Johan Fredborg

Sammanfattning
Abstract

Communication through text messaging, SMS (Short Message Service), is nowadays a huge industry with billions of active users. Because of the huge user base it has attracted many companies trying to market themselves through unsolicited messages in this medium in the same way as was previously done through email. This is such a common phenomenon that SMS spam has now become a plague in many countries.

This report evaluates several established machine learning algorithms to see how well they can be applied to the problem of filtering unsolicited SMS messages. Each filter is mainly evaluated by analyzing the accuracy of the filters on stored message data. The report also discusses and compares requirements for hardware versus performance measured by how many messages that can be evaluated in a fixed amount of time.

The results from the evaluation shows that a decision tree filter is the best choice of the filters evaluated. It has the highest accuracy as well as a high enough process rate of messages to be applicable. The decision tree filter which was found to be the most suitable for the task in this environment has been implemented. The accuracy in this new implementation is shown to be as high as the implementation used for the evaluation of this filter.

Though the decision tree filter is shown to be the best choice of the filters evaluated it turned out the accuracy is not high enough to meet the specified requirements. It however shows promising results for further testing in this area by using improved methods on the best performing algorithms.

Nyckelord
Keywords Spam filtering, Machine Learning, C45, Support Vector Machine, Dynamic Markov Coding, Nave Bayes