Logistic Regression and Spam Filtering

Duncan Findlay
Steven Birk

April 9, 2007

Supervisors
Dr. Glen Takahara
Dr. Tamas Linder
Acknowledgements

We would like to thank the following people for their assistance with this project.

- Dr. Glen Takahara
- Dr. Tamas Linder
- Dr. Andrew Ng and Jeremy Kolter, Stanford University
- Dr. Paul Komarek and Dr. Andrew W. Moore, Carnegie Mellon University
- The developers of the Apache SpamAssassin project
Abstract

Apache SpamAssassin is an open-source rule-based spam filter. The filter’s performance is highly dependent on how scores are generated for the rules that are used. Our project was to investigate the use of logistic regression as a technique for generating these scores. We examined two different algorithms to perform the logistic regression calculations, and found that the TR-IRLS algorithm worked best for our purposes. The SpamAssassin scores we generated with this algorithm are better in some situations than the scores generated by the Genetic Algorithm (GA), the method that is currently used by the SpamAssassin project. One major benefit with the TR-IRLS algorithm is that it is very fast compared to the GA, which takes hours to run. With more work to improve the scoring process using TR-IRLS, it should be possible to develop a replacement for the GA that is superior in all situations.
Chapter 1

Introduction

E-mail is an essential form of communication in our digital world. Unfortunately, it is often abused by people wishing to use it for advertising their products. Unsolicited bulk e-mail, better known to users as “spam” is a huge problem. One estimate suggests that American companies are losing $71-billion per year in productivity due to spam.[6]

One of the most widely used spam filters, Apache SpamAssassin\textsuperscript{TM}\footnote{SpamAssassin\textsuperscript{TM} is a trademark of the Apache Software Foundation} is developed as an open-source project by people all across the world. SpamAssassin works by searching through an e-mail message for certain patterns or characteristics of a message, specified as “rules”. Each rule has an associated score, and the scores for each rule that matched on a message are summed. If this total score is greater than a threshold value, the message is marked as spam, otherwise it is marked as non-spam (often referred to as “ham” in the anti-spam world). Choosing the proper scores for each rule is essential for the accuracy of the software.

One of the biggest challenges for the Apache SpamAssassin project is to optimize the scores for each rule. The scores must be chosen so that spam messages have a total score for the message that is greater than or equal to the default threshold of 5.0, and non-spam messages have a smaller score.

In order to optimize scores, volunteers keep collections of old spam and ham e-mails they have received. Before a major release of SpamAssassin, these volunteers scan all of these messages with SpamAssassin and keep logs of which rules hit on which message. (This is an automated process, so it is not as tedious as it sounds!)
This e-mail data can then be fed into an algorithm to generate scores. The project currently uses a custom-made algorithm known as the Genetic Algorithm (GA) to optimize scores. It takes hours to run in order to generate scores. In the past, a much faster algorithm, known as the Perceptron, was used, but the implementation is currently broken and unable to produce good scores.

This project focuses on the method used to generate the scores for SpamAssassin. While the current score generation method is good, there is lots of room for improvement. No spam filter is perfect; every filter will have false negatives (spam messages not caught by the filter) and false positives (legitimate messages marked as spam). If we improve the score generation method, we can reduce the number of misclassifications, leading to better performance.

In July 2005, at an informal gathering after the Conference on E-mail and Anti-Spam, Dr. Andrew Ng, Assistant Professor of Computer Science at Stanford University suggested to the developers of SpamAssassin that it may be beneficial to use $L_1$ regularized logistic regression to determine if a message is spam. This lead us to choose this project.

The purpose of this project is to investigate the use of logistic regression as a score generation technique for Apache SpamAssassin.
Chapter 2

Background and Theory

2.1 Logistic Regression

Logistic regression is a statistical model that assumes the probability of an object being in a class is distributed according to the formula shown in Equation 2.1. In our application an object is an e-mail message, and it can be in one of two classes, either ham or spam.

\[
Pr(y = 1|x; \theta) = \mu(x, \theta) = \frac{1}{1 + e^{-\theta^T x}}
\]  

(2.1)

We are trying to determine the probability that a message is spam \((y = 1)\) given the rules that matched on the message. The feature vector \(x \in \{0, 1\}^N\) represents which of the \(N\) rules are matched on a given message \((x_i = 1\) if the \(i\)th rule matched on the message). The goal is to find the weights vector \(\theta \in \mathbb{R}^N\) that results in Equation 2.1 being a good estimate of the probability.

We use maximum likelihood estimation to find \(\theta\). To do this, we use a training set of \(M\) training instances \(\{(x^{(i)}, y^{(i)}), i = 1, \ldots, M\}\), where each \(x^{(i)} \in \mathbb{R}^N\) is the feature vector, and \(y^{(i)}\) is the class of the message. For a given value of \(\theta\), we have the likelihood function \(L(\theta)\) defined below.

\[
L(\theta) = \prod_{i=1}^{M} Pr(Y = y^{(i)}|x^{(i)}; \theta)
\]

(2.2)

We want to find the value of \(\theta\) that maximizes the likelihood function. Equivalently, we can find a value of \(\theta\) that maximizes the log-likelihood function \(l(\theta)\), shown below.
\[ l(\theta) = \log(L(\theta)) \]  
\[ = \sum_{i=1}^{M} \log \Pr(y^{(i)}|\mathbf{x}^{(i)}; \theta) \]  
\[ \theta = \arg \max_{\theta} \sum_{i=1}^{M} \log \Pr(y^{(i)}|\mathbf{x}^{(i)}; \theta) \]  

We need to find \( \theta \) as shown in Equation 2.5.

### 2.2 Algorithms

In order to use logistic regression, we explored two algorithms for determining \( \theta \) as in Equation 2.5. The first algorithm was IRLS-LARS developed by Lee et al.\[5\], designed for \( L_1 \) regularized logistic regression. The second algorithm was the TR-IRLS algorithm developed by Komarek and Moore\[8\], designed for binary classification on large binary datasets.

#### 2.2.1 IRLS

Both algorithms we explored used some form of the IRLS (Iteratively Reweighted Least Squares) algorithm in order to search for the maximum likelihood estimate of the logistic regression parameter \( \theta \). The problem of finding the maximum likelihood estimate shown in Equation 2.5 may be expressed as the equivalent optimization problem shown in Equation 2.6.

\[ \theta = \arg \min_{\theta} \sum_{i=1}^{M} - \log \Pr(y^{(i)}|\mathbf{x}^{(i)}; \theta) \]  

The IRLS algorithm is a iterative, non-linear optimization algorithm which determines the solution to Equation 2.6 through a series of weighted least squares subproblems. Let \( X \in \mathbb{R}^{M \times N} \) denote the matrix with each row representing a feature vector \( \mathbf{x}^{(i)} \in \mathbb{R}^{N} \). Let \( y^{(i)} \) denote the class corresponding to the \( i \)th feature vector. Let \( \theta^{(k)} \) represent the value of \( \theta \) on the \( k \)th iteration of the IRLS algorithm. The algorithm takes as input an initial estimate of the regression parameter \( \theta^{(0)} \), the training data \( X \), and the
vector $\mathbf{y} = (y^{(1)}, \ldots, y^{(N)})$. The algorithm uses the estimate of $\theta$ from the current iteration to form a linear system of equations known as a weighted least squares subproblem. This subproblem is solved to produce the next estimate of the parameter. The algorithm iterates through this process until the estimate of $\theta$ converges.

For the $k$th iteration a diagonal weights matrix $W$ is computed, which has entries $W_{ii}$ as follows for $i = 1, 2, \ldots, M$:

$$W_{ii} = \mu(x^{(i)}, \theta^{(k)}) (1 - \mu(x^{(i)}, \theta^{(k)})) \tag{2.7}$$

Also, the vector $\mathbf{z} \in \mathbb{R}^M$ is computed, where each entry is as follows for $i = 1, 2, \ldots, M$:

$$z_i = (\theta^{(k)})^\top x^{(i)} + \frac{(y^{(i)} - \mu(x^{(i)}, \theta^{(k)}))}{W_{ii}} \tag{2.8}$$

The linear system representing the weighted least squares subproblem for the $k$th iteration to determine $\theta^{(k+1)}$ then becomes:

$$(X^\top W X)\theta^{(k+1)} = X^\top W z \tag{2.9}$$

Each subproblem is a linear system of equations that can be solved in many different ways. Each iteration of the IRLS algorithm requires finding the solution to a new subproblem (since the weights matrix changes on each iteration), so the performance of the IRLS algorithm depends greatly on which approach is used. The two algorithms we explored used different approaches to solve the weighted least squares subproblems. The specific approaches used are discussed in Sections 2.2.2 and 2.2.3.

For further information about the IRLS algorithm, see [4].

2.2.2 IRLS-LARS

The IRLS-LARS (Iteratively Re-weighted Least Squares Least Angle Regression) algorithm was developed by Lee et al. [5] to perform $L_1$ regularized logistic regression. This form of regression tries to find a solution to the
following optimization problem:

$$
\min_{\theta} \sum_{i=1}^{M} - \log p(y^{(i)} | x^{(i)}; \theta)
$$

subject to $\|\theta\|_1 \leq C$.

(2.10)

The IRLS-LARS algorithm uses IRLS to search for the maximum likelihood estimate while subjecting the solution of each weighted least squares subproblem to the $L_1$ constraint shown in Equation (2.10). For further explanation of this modification please refer to [5].

In order to solve each $L_1$ constrained least squares subproblem, the IRLS-LARS algorithm uses the LARS (Least Angle Regression) algorithm with the LASSO modification. The LARS algorithm is discussed in more detail in [3].

For further information regarding convergence guarantees, as well as performance analysis of the IRLS-LARS algorithm please refer to [5].

### 2.2.3 TR-IRLS

The TR-IRLS (Truncated Regularized Iteratively Reweighted Least Squares) algorithm was developed by Komarek and Moore. It was designed for use as a data mining tool, and therefore scales well to very large sparse data sets. It is also designed with binary classification in mind.

Similar to the IRLS-LARS algorithm, TR-IRLS uses IRLS to search for the maximum likelihood estimate of the logistic regression parameter $\theta$. However, at each stage of IRLS, it uses a different technique to solve the weighted least squares subproblem (this is not the constrained version as used in IRLS-LARS). Since each weighted least squares subproblem is a linear system of equations, an iterative minimization algorithm known as the Conjugate Gradient method is used. The use of the Conjugate Gradient method is pivotal in the algorithm’s scalability as the runtime of this method is $O(n)$ where $n$ is the number of non-zero entries in the linear system. The TR-IRLS algorithm has been designed for very large sparse binary data sets, which contain few non-zero entries. For an in-depth description of the Conjugate Gradient method, please see [9]. A pseudo-code description of the TR-IRLS algorithm may be seen in Figure 2.1.

The Conjugate Gradient method used to solve each least squares subproblem is stopped early (“truncated”) to provide an estimate to the solution of
CHAPTER 2. BACKGROUND AND THEORY

the subproblem. The truncation of the Conjugate Gradient method helps to speed up each iteration of the IRLS algorithm. As a result the TR-IRLS algorithm runs very quickly. Ridge regression (which is a form of Tikhonov Regularization) is used to address issues of correlated attributes that arise from the use of very large data sets. Correlated attributes cause problems as rows of the data matrix may become linear combinations of each other, sometimes making a unique solution to the weighted least squares subproblem impossible. Ridge regression addresses this by modifying the system of linear equations in order to estimate the solution to ill-posed (ill-conditioned) subproblems. Further information about the ridge regression method may be found in [7].

\[
\begin{align*}
\text{set } i &= 0 \\
\text{repeat} & \\
\quad \text{compute } \mu(\mathbf{x}^{(j)}, \theta^{(i)}) \text{ for } j = 1, \ldots, M \\
\quad \text{compute weights matrix } W = \text{diag}(w_1, \ldots, w_M) \\
\quad \quad \text{where } w_j = \mu(\mathbf{x}^{(j)}, \theta^{(i)})(1 - \mu(\mathbf{x}^{(j)}, \theta^{(i)})) \text{ for } j = 1, \ldots, M \\
\quad \text{compute } \mathbf{z} \text{ as shown in Section 2.2.1} \\
\quad \text{formulate weighted least squares subproblem:} \\
\quad (X^\top W X) \theta^{(i+1)} &= X^\top W \mathbf{z} \\
\quad \text{compute } \theta^{(i+1)} \text{ from WLS using truncated Conjugate Gradient method} \\
\quad i &= i + 1 \\
\text{until } & \theta^{(i+1)} \text{ converges}
\end{align*}
\]

Figure 2.1: Pseudocode for the TR-IRLS algorithm

Further explanation of the TR-IRLS algorithm, optimizations used, as well as performance analysis may be seen in [8].

2.3 Performance Analysis

2.3.1 Total Cost Ratio

The Total Cost Ratio (TCR) is a metric often used by the SpamAssassin project [2], and was first developed by Androutsopoulos et al. [11]. The TCR is
the ratio of the cost of manually sorting incoming e-mail compared to the cost of cleaning up any errors made by the spam filter. This second cost includes the cost of getting rid of spam messages marked as ham (false negatives), and to recover from ham messages marked as spam (false positives).

It is more costly to recover from false positives than false negatives, so the TCR depends on a parameter $\lambda$, which represents the cost of a false positive relative to a false negative. Different values of $\lambda$ should be chosen based on how the user of the software deals with messages marked as spam. The SpamAssassin project uses $\lambda = 50$ which is supposed to represent the cost of recovering false positives from an e-mail folder which is rarely checked.

The TCR is calculated as shown in Equation (2.11), where $N_s$ is the number of spam messages, $N_{fp}$ is the number of false positives (ham identified as spam) and $N_{fn}$ is the number of false negatives (spam identified as ham).

$$TCR = \frac{N_s}{\lambda \cdot N_{fp} + N_{fn}}$$ (2.11)

A greater TCR means that the spam filter is more accurate. A value of 1 indicates that the filter provides absolutely no benefit to the user, and a value less than 1 indicates that the spam filter actually inconveniences the user.

### 2.3.2 Receiver Operating Characteristic

A Receiver Operating Characteristic (ROC) is a graph that shows the ability for a binary classifier with a real-valued output to correctly classify objects into two classes as the threshold value is changed. The plot shows the number of false positives on the x-axis and the number of true positives on the y-axis. Every point on the curve indicates the tradeoff between true positives and false positives at a particular threshold value.

To create a ROC curve, the objects (in our case, e-mail messages) are sorted by their score in descending order. Starting in the bottom left corner, the curve goes up one unit if the message is in the positive class (i.e. spam) and right one unit if the message is in the negative class. The curve always ends up in the top right corner of the graph (with coordinates $(N_1, N_0)$, where $N_1$ is the number of objects in the positive class, and $N_0$ is the number in the negative class). By changing the scale of the axes, we can instead plot percentages of messages rather than actual numbers of messages in each class.
2.4 Apache SpamAssassin

2.4.1 Rules

Apache SpamAssassin is a rule-based filter. This means it has a collection of “rules” that it looks for in an e-mail message. Most of these rules are simple patterns that are defined by a regular expression that can be matched against the headers of the e-mail, a URL found in the message, the message body or the whole e-mail. A simple rule definition is shown in Listing 2.1. These rules are defined in simple configuration files, so users can easily add new rules. The first line is the definition of the test in terms of a regular expression. The second line is a human readable description that can be shown to users, and the third is the scores that will be used for it. (There are four scores corresponding to four score sets as explained in the next section.)

Some rules (known as “eval rules”) are defined by Perl subroutines that return a positive value if the rule matches and 0 otherwise. Often these are fairly simple, but there is no limit to how powerful they can be.

For example, there are a set of rules that hook a naïve Bayesian Classifier into SpamAssassin. This system attempts to predict the probability of a message being spam based on the words it contains and the past occurrences of these words in spam and ham. If this system is turned on, the probability for each message is calculated, and one of nine “Bayes” rules will then match the message depending on which range this probability is in. Since the Bayesian Classifier needs to be trained, and it needs to store its data somewhere, it is disabled in many configurations.

There are also a number of rules that query remote blacklists of known
IP addresses that are sources of spam, and blacklists of URLs that have been advertised in spam. These are known as the “network rules”. These are sometimes disabled by system administrations since they are slow; the SpamAssassin engine must wait for the remote hosts to return the answer to its query before it can determine if a message is spam.

2.4.2 Score Sets

As mentioned above, there are a number of sets of rules that are often disabled by system administrators. Both the Bayes rules and the network rules are very good and often add a few points to a message’s score. It makes sense that the “regular” rules should be assigned larger scores when these rules are disabled, so the SpamAssassin project actually uses four sets of scores to correspond to the four different configurations that can be used. Each score set is identified by a number, as shown in Figure 2.2. This means we need to generate four sets of scores.

<table>
<thead>
<tr>
<th>Network Tests</th>
<th>Bayes Disabled</th>
<th>Bayes Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Enabled</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 2.2: Score set numbering

2.5 Practical Considerations

There are a number of additional factors that must be taken into account when generating scores, which are not found in most machine learning problems.

Because SpamAssassin is an open-source project, the scores assigned to each rule are publicly visible. As a result, spammers can carefully craft their e-mail messages to avoid or intentionally hit certain rules. Thus, it is necessary to ensure that rules are not given negative scores unless there is no feasible way to exploit the negatively scoring rule. (Some of the SpamAssassin rules, for example rules designed to “whitelist” certain senders, are indeed scored negative, but spammers have no way of faking their messages
CHAPTER 2. BACKGROUND AND THEORY

so that these rules hit.) Specifically, we must ensure that rules that are
designed to match spam do not get assigned a negative score.

While the SpamAssassin project strives to get a representative sample
of real-world e-mail, this is difficult as every user’s e-mail is different. For
example, the word “Viagra” might be a strong sign of ham for an executive at
Pfizer Inc., the corporation that produces Viagra; however, for your average
computer user, it is a strong sign that a message is spam. As a result, the
project has set limits on what is an acceptable score for each rule. Generally,
no rule should be given a score of more than 4.5 (the default threshold is
5.0), so that no one rule can be responsible for blocking a message. Rules
are given a smaller maximum score based on their performance against the
training data; if a rule (designed to catch spam) matches too many ham
messages or too few spam messages, its maximum score is decreased. These
limits are known as “score ranges” (and are generated by a script called
score-ranges-from-freqs).

In addition to the above restrictions, some rules have hard-coded scores
that are not to be automatically generated. These rules are known as “im-
mutable rules”. For example, some of the rules designed to “whitelist” cer-
tain senders based on a remote database have hard-coded negative scores,
since these rules do not match on many messages (and as a result would be
given a very small “score range”). They are known to be very accurate, so the
score generation process is bypassed and they are assigned immutable scores.
Other immutable rules are given very small positive scores as they are in-
tended to be “informational”. For example the “ANY_BOUNCE_MESSAGE”
rule is given a score of 0.1 so that users can filter e-mail bounce messages
into a separate folder based on whether that rule hit.

The “Bayes” rules are also immutable. The logic behind this is question-
able, but when they were mutable, they were assigned scores that did not
really make sense. “BAYES_99,” (the rule which indicates the probability of
the message being spam according to the Bayesian classifier is greater than
99%) was assigned a score that was less than that assigned to “BAYES_95”
(which indicates the probability is between 95% and 99%). This was counter-
intuitive, and the developers decided that these scores should be hard-coded
to logical values.

Rules also get marked as “immutable,” and have their scores set to 0, if
they are not found in the training set in sufficient quantity. This essentially
disables the rule, and the SpamAssassin software will not check a message
for a rule with a score of 0.
2.6 Current Score Generation Methods

The algorithms currently used by the SpamAssassin project for score generation are known as the Perceptron, and the Genetic Algorithm (GA).

The Perceptron uses the stochastic gradient descent method for training a neural network. It uses a basic neural network with a linear transfer function, and a logarithmic sigmoid activation function. The Perceptron is currently unable to produce good scores due to technical problems in its implementation.

The Genetic Algorithm is an implementation of a stochastic search method where possible solutions (sets of SpamAssassin scores) are represented as objects called “genomes” and through a series of “mutations” and “mating” of these objects, the best solution is determined using a set of selection criteria.

Both algorithms used by Apache SpamAssassin have been specifically designed to handle score ranges and immutable rules as described in the previous section. Further resources on the Perceptron and the GA may be found at [2].
Chapter 3
Design

3.1 SpamAssassin Integration

Regardless of the algorithm used, in order to use logistic regression with SpamAssassin, we have two options. One option is to modify the SpamAssassin engine in order to use logistic regression weights as scores and mark messages as spam if the probability threshold is exceeded. We would need to modify the logic for determining the total message score; this is quite a lot of work, for relatively little gain.

The other option is to convert the logistic regression weights into scores that are suitable for SpamAssassin, given a probability threshold. This is trivial. Given a vector of scores $s \in \mathbb{R}^N$ and a feature vector $x \in \mathbb{R}^N$, SpamAssassin marks a message as spam if the following inequality is satisfied.

$$s^\top x > 5$$  \hfill (3.1)

We would like to instead use the following decision rule, which can be re-arranged as shown.

$$\Pr(Y = 1|x, \theta) > t$$  \hfill (3.2)

$$\frac{1}{1 + e^{-\theta^\top x}} > t$$  \hfill (3.3)

$$\theta^\top x > -\ln \frac{1 - t}{t}$$  \hfill (3.4)

Equations 3.2 through 3.4 demonstrate how the linear regression weights can be converted to linear weights given a probability threshold $t$. The
logistic regression model predicts a message is spam when Equation 3.2 is satisfied. Hence, as shown in Equation 3.4 we can simply use these weights as SpamAssassin scores if we set the threshold to $-\ln \frac{1-t}{t}$.

Ideally, we would like to keep our default threshold of 5 for SpamAssassin (to avoid having to re-write the documentation, for example). So, we can scale these weights by a scalar $c$, to input to SpamAssassin with no modifications necessary. This is shown in Equations 3.5 to 3.7.

$$c = \frac{5}{-\ln \frac{1-t}{t}}$$ \hspace{1cm} (3.5)

$$c\theta^\top x > c \left(-\ln \frac{1-t}{t}\right)$$ \hspace{1cm} (3.6)

$$(c\theta)^\top x > 5$$ \hspace{1cm} (3.7)

In reality, it is slightly more complicated because both of the logistic regression algorithms actually consider a feature vector $\hat{x} = (1, x)$ instead of just $x$ (i.e. they include a “dummy feature” that is 1 for all of the objects), and return $\hat{\theta} = (b, \tilde{\theta})$. Essentially this means that there is a scalar $b$ such that $\hat{\theta}^\top \hat{x} = \tilde{\theta}^\top x + b$. We define $c$ slightly differently to take this into account, as shown in Equations 3.8 to 3.11.

$$c = \frac{5}{-\ln \frac{1-t}{t} - b}$$ \hspace{1cm} (3.8)

$$\hat{\theta}^\top \hat{x} > -\ln \frac{1-t}{t}$$ \hspace{1cm} (3.9)

$$\tilde{\theta}^\top x > -\ln \frac{1-t}{t} - b$$ \hspace{1cm} (3.10)

$$(c\tilde{\theta})^\top x > 5$$ \hspace{1cm} (3.11)

In essence, this means that if we develop a successful score generation technique using logistic regression, it can be a drop-in replacement for the current systems.

### 3.2 Data Collection and Formatting

In order to generate scores for Apache SpamAssassin, we need a large training set of data. The SpamAssassin project already has a process to collect these
data, and we were fortunate to be able to use the data collected for the upcoming 3.2.0 release of their software.

Volunteers worldwide run a special script known as \textit{mass-check} on their collections of known ham and spam e-mail. This script uses the SpamAssassin libraries to test each message, and it will output a single line for each message as shown in Listing 3.1. Each line contains (in order) whether the message is determined to be spam (either “Y” or “.”), the score the message is given based on current scores (rounded to the nearest integer), the path to the message location on the local machine, a list of rules hit separated by commas, and then some extra data (in \textit{key=value} format). Each of these 5 fields is separated by a space.

Listing 3.1: Sample line of output of \textit{mass-check}

\begin{verbatim}
Y 13 /home/duncf/Maildir/Old/spam/spam200612.87396010 AWL, FH_HOST_EQ_D,D,D,D,FH_HOST_EQ_D,D,DB,FH_MSGID_01C67, FRT_PENIS1,HDR_ORDER_FTSDMCMXX_001C,HELO_LOCALHOST, RCVD_IN_SORBS_DUL,T_RCVD_IN_SORBS_DUL,T_SURBL_MULTI1, T_SURBL_MULTI2,T_URIBL_RHS_URIBL_BLACK,URIBL_JP_SURBL, URIBL_SC_SURBL,URIBL_WS_SURBL,_ANY_OUTLOOK_MUA,_CT, _CTYPE,CHARSET_QUOTED,_CT_TEXT_PLAIN,_DOS_RCVD_WED, _ENV_AND_HDR_FROM_MATCH,_FB_S_PRICE,_FH_HAS_XMSMAIL, _FH_HAS_XPRIORIY,_FH_MSGID_00001C,_FH_MSGID_01C7, _FM_MY_PRICE,_HAS_ANY_URI,_HAS_MIMEOLE,_HAS_MSGID, _HAS_MSMAIL_PRI,_HAS_RCVD,_HAS_SUBJECT,_HAS_X_MAILER, _HDR_ORDER_FTSDMCXXX,_HELO_NO_DOMAIN, _LOCAL_PP_NONPPURL, _MID_START_001C, _MIMEOLE_MS, _MIME_VERSION,_MISSING_REF,_MISSING_REPLY, _MISSING_THREAD,_MSGID_DOLLARS_MAYBE, _MSGID_DOLLARS_OK,_MSGID_OK_HEX,_MSGID_OK_HOST, _MSGID_RANDY,_MSGID_VGA, _NONEPTY_BODY, _NO_INR_YES_REF,_OE_MSGID_2,_OE_MUA,_RCVD_IN_SORBS, _SANE_MSGID,_TOCC_EXISTS,_XM_MSOE6,_XM_MS_IN_GENERAL, _XM_OUTLOOK_EXPRESS time=1167211856, scantime=4, format=m, reuse=no
\end{verbatim}

These \textit{mass-check} ham and spam logs are each concatenated and then they are input into the scoring algorithm. These logs are our raw data. For the Perceptron and the Genetic Algorithm (GA), there is a collection
of Perl scripts that convert these data into C code that can be compiled into the algorithm. The most important of these scripts is called *logs-to-c*. In order to evaluate any other scoring mechanism, it is necessary to modify this script to output data in the format needed for our analysis.

One important task of the *logs-to-c* script is to take messages which hit the same set of rules and combine them into one data entry, (possibly) with an increased weight. This is often necessary in machine learning problems as re-learning the same training sample provides no new information and provides no benefit.

The raw data used in the evaluation of our algorithms were provided by the Apache SpamAssassin project. These are the same data that were used to generate scores for their official releases, and the most recent data set, collected for the upcoming 3.2.0 release of SpamAssassin has results from 1,681,081 messages.

Before using the data to train a model, the data are split into two sets; the training set consists of 90% of the data and the validation set consists of the other 10%. The training set is used for the score generation, while the validation set is used only to evaluate the quality of the scores. If we evaluated our results on the data used to train our model, we would expect to obtain much higher accuracy, which is not representative of real-world performance.

### 3.3 IRLS-LARS

#### 3.3.1 Implementation

Jeremy Kolter, a graduate student working with Dr. Andrew Ng at Stanford University was able to provide us with Matlab code of the IRLS-LARS algorithm from which we could begin our analysis.

The first task was to convert our *mass-check* logs into a format that could be imported into Matlab. We modified the *logs-to-c* script to output the data into a text representation of large matrix. (Our resulting script is called *logs-to-matlab*.)

In order to evaluate the IRLS-LARS algorithm as a possible scoring mechanism, we developed a number of Matlab functions to assist with the analysis of the data. We modified the provided Matlab code somewhat to suit our data, and we developed code to take the results from the algorithm and test
its performance against validation data. The functions used to carry out our testing, validation and evaluation process are shown in Figure 3.1.

The IRLS-LARS algorithm takes a parameter $C$, which represents the constraint on the weights vector, $\theta$, as described in Section 2.2.2. Since our model gives a probability of a message being spam, we pick a threshold value $t$, such that all messages with probability of spam greater than $t$ are considered spam, and messages with a smaller probability are considered ham. In order to compare the accuracy of the model for different values of $C$ and $t$, we used Total Cost Ratio, as described in Section 2.3.1.

We can search for the optimal values for $t$ and $C$ based on the TCR of the model on the training set.

More work would need to be done to actually take the scores generated by this algorithm and convert them to SpamAssassin scores. This was not done, for reasons that will be explained in Chapter 5.
3.4 TR-IRLS

Dr. Paul Komarek and Dr. Andrew Moore of Carnegie Mellon University have released software to perform TR-IRLS under the GNU General Public License, as a package called lr_trirls. It is this software that we have used for our analysis. There is one main program that we use, train, which takes a sparse matrix representation of the data as input, and outputs the weights vector $\theta$.

3.4.1 Initial Training Process

To begin, it was necessary for us to modify the logs-to-c to output the data in the sparse matrix representation required by the lr_trirls software. Our new script, known as logs-to-trirls, outputs a file with lines as shown in Listing 3.2. Each line represents a message. The first column contains the message class, and the subsequent columns indicate which rules are hit by the message (this is a compact way of writing a sparse matrix). It also outputs a file listing the names of the rules in numerical order (so we can map from number to rule).

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>117</td>
<td>380</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>224</td>
<td>231</td>
<td>449</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>213</td>
<td>224</td>
<td>239</td>
<td>351</td>
</tr>
<tr>
<td>1</td>
<td>380</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>224</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The program train is called and it returns a file containing the parameter weights (the $\theta$ vector) and the number of rules. This data is combined with the rule names file generated by logs-to-trirls to create a file called trirls.scores, which contains a list of parameter weights and the rule they represent. This entire process is shown in Figure 3.2.
3.4.2 Probability Threshold Selection

Once we have the parameter weights for each rule, we can try to determine which probability threshold we should use to separate ham and spam to maximize our accuracy. To do this, we compare the accuracy of different probability thresholds using Total Cost Ratio, as described in Section 2.3.1.

We wrote a few scripts to help automate this process. `evaluate-lr` uses the training data and the `trirls.scores` file to output a file containing one line for each message with the message class and the probability the message is spam according to the model. `evaluate-lr-threshold` takes a probability threshold, \( t \), as a parameter. It uses the output of `evaluate-lr` to compute statistics about the accuracy of the model for that threshold. `optimize-lr-threshold` runs `evaluate-lr` on various values of \( t \) in an attempt to find the threshold that yields the highest TCR.

This is summarized in Figure 3.3.

3.4.3 Converting to SpamAssassin Scores

As explained in Section 3.1, it is trivial to convert from logistic regression weights to SpamAssassin scores. Unfortunately, when we try to do this,
we end up with scores outside the permitted score ranges described in Section 2.5. The TR-IRLS algorithm does not differentiate between mutable and immutable rules, and has no understanding of score ranges. (We do not generate scores for rules that are not enabled in a given score set, but that is the only restriction on which scores are generated.)

Instead, if our score ends up being outside of the allowable range for the rule, we adjust it to the nearest allowable score. So, for example, suppose a rule is permitted to have a score between 0 and 4.5. If the logistic regression weight scales to be 5.6, we simply set the score to 4.5.

This work is all done by the trirls-to-spamassassin script we wrote. It takes as input the probability threshold to use (to determine the scaling factor), the trirls.scores file containing the parameter weight for each rule and the output of score-ranges-from-freqs, the script the SpamAssassin project uses to generate score ranges.

trirls-to-spamassassin outputs a SpamAssassin configuration file that contains the scores for one score set. The SpamAssassin project script rewrite-cf-with-new-scores rewrites the main scores file with the new scores for the
appropriate score set.

This process is illustrated in Figure 3.4

3.4.4 Iterating

While we had initially assumed that the optimal probability threshold calculated by the `optimize-lr-threshold` script would be optimal once the scores were converted to SpamAssassin scores, this turned out to be false.

The adjustments made to the scores to make them fit the allowable score ranges greatly affected the accuracy of the model. After updating the SpamAssassin scores, we used SpamAssassin’s `mk-roc-graphs` script\(^1\) to output the TCR on the training data at every SpamAssassin threshold. By determining

\(^1\)This script has a misleading name, as it is designed to make ROC curves, but has also been adapted to simply output the TCR at all possible thresholds (in increments of .1), when given the `–count` option.
which SpamAssassin threshold yields the highest TCR, we can adjust the probability threshold used to scale the scores in order to get the optimal threshold to be 5.0. It is important to note that this optimization needs to be done using the training data, otherwise we are “cheating” by using our validation set in the score generation process. This process is shown in Figure 3.5.

If the TCR is optimal for a SpamAssassin threshold less than 5.0, we need to decrease our probability threshold given to trirls-to-spamassassin (at 5.0 we are marking too many messages as ham). If the optimal SpamAssassin threshold is greater than 5.0 we need to increase our probability threshold. If we change our probability threshold, the scores are updated for this new threshold, as shown in the previous section.

Unfortunately, this is a manual, iterative step. A future improvement would be to make this step automated.

### 3.4.5 Validation

Once we have optimized our scores on the training data, we can evaluate our performance on the validation set.

To do this, we use the SpamAssassin project’s script *fp-fn-statistics* with the validation set to generate statistics for the scores at the threshold 5.0.
This is shown in Figure 3.6. The script *parse-rules-for-masses* is simply a helper script that the SpamAssassin project uses to parse the rules and scores from the configuration files for their score generation scripts.
Chapter 4

Results

In this chapter we present performance results for the scores generated by the SpamAssassin project using the Genetic Algorithm and the scores we generated using the TR-IRLS algorithm and the scoring process described in Chapter 3.

We were unable to produce reasonable results using the IRLS-LARS algorithm due to various limitations which are discussed in Section 5.1. In addition, results are not shown for the scores generated by the Perceptron as the implementation is currently broken.

4.1 Statistics

The SpamAssassin project’s script \textit{fp-fn-statistics} was used to generate the results shown below. This script calculates the following values:

- **Correctly non-spam** - indicates how many non-spam messages were correctly marked as non-spam (ham), as a number and as a percentage of all non-spam messages.

- **Correctly spam** - indicates how many spam messages were correctly marked as spam, as a number and as a percentage of all spam messages.

- **False positives** - indicates how many ham messages were incorrectly marked as spam, as a number and as a percentage of all ham messages.

- **False negatives** - indicates how many spam messages were incorrectly marked as ham, as a number and as a percentage of all spam messages.
CHAPTER 4. RESULTS

- **TCR** - the total cost ratio, as described in Section 2.3.1, calculated using a $\lambda$ value of 50

- **SpamRecall** - Spam Recall, of all the spam messages, the percentage of messages that were correctly marked as spam

- **SpamPrec** - Spam Precision, of all the messages marked as spam, the percentage of messages that were actually spam

The outputs of this script for each set of scores and for both algorithms are shown below in Listings 4.1-4.8. These results were generated using the validation set of data, not the training set.

Listing 4.1: TR-IRLS results for score set 0

```
# Correctly non-spam: 67444 99.84%
# Correctly spam: 103993 87.33%
# False positives: 106 0.16%
# False negatives: 15090 12.67%
# TCR(\(l = 50\)) : 5.840265 SpamRecall : 87.328% SpamPrec : 99.898%
```

Listing 4.2: GA results for score set 0

```
# Correctly non-spam: 67074 99.30%
# Correctly spam: 109992 92.37%
# False positives: 476 0.70%
# False negatives: 9091 7.63%
# TCR(\(l = 50\)) : 3.620534 SpamRecall : 92.366% SpamPrec : 99.569%
```

Listing 4.3: TR-IRLS results for score set 1

```
# Correctly non-spam: 67495 99.92%
# Correctly spam: 110019 92.39%
# False positives: 55 0.08%
# False negatives: 9064 7.61%
# TCR(\(l = 50\)) : 10.079821 SpamRecall : 92.389% SpamPrec : 99.950%
```

Listing 4.4: GA results for score set 1

```
# Correctly non-spam: 67386 99.76%
# Correctly spam: 114216 95.91%
```
### CHAPTER 4. RESULTS

<table>
<thead>
<tr>
<th># False positives: 164 0.24%</th>
<th># False negatives: 4867 4.09%</th>
<th># TCR(1=50): 9.113262 SpamRecall: 95.913% SpamPrec: 99.857%</th>
</tr>
</thead>
</table>

**Listing 4.5: TR-IRLS results for score set 2**

<table>
<thead>
<tr>
<th># Correctly non-spam: 67487 99.91%</th>
<th># Correctly spam: 114242 95.93%</th>
<th># False positives: 63 0.09%</th>
<th># False negatives: 4841 4.07%</th>
<th># TCR(1=50): 14.902140 SpamRecall: 95.935% SpamPrec: 99.945%</th>
</tr>
</thead>
</table>

**Listing 4.6: GA results for score set 2**

<table>
<thead>
<tr>
<th># Correctly non-spam: 67507 99.94%</th>
<th># Correctly spam: 114861 96.45%</th>
<th># False positives: 43 0.06%</th>
<th># False negatives: 4222 3.55%</th>
<th># TCR(1=50): 18.688481 SpamRecall: 96.455% SpamPrec: 99.963%</th>
</tr>
</thead>
</table>

**Listing 4.7: TR-IRLS results for score set 3**

<table>
<thead>
<tr>
<th># Correctly non-spam: 67520 99.96%</th>
<th># Correctly spam: 116532 97.86%</th>
<th># False positives: 30 0.04%</th>
<th># False negatives: 2551 2.14%</th>
<th># TCR(1=50): 29.395952 SpamRecall: 97.858% SpamPrec: 99.974%</th>
</tr>
</thead>
</table>

**Listing 4.8: GA results for score set 3**

| # Correctly non-spam: 67508 99.94% | # Correctly spam: 117293 98.50% | # False positives: 42 0.06% | # False negatives: 1790 1.50% | # TCR(1=50): 30.612596 SpamRecall: 98.497% SpamPrec: 99.964% |
4.2 ROC Graphs

Figures 4.1 and 4.2 show ROC curves for score set 0, as described in Section 2.3.2. Figure 4.1 shows a plot of the entire ROC graph, while Figure 4.2 focuses on the area of interest. On these plots the “Straight LR” curve was generated using the logistic regression model and the weights generated by the TR-IRLS algorithm. The “LR generated SA scores” curve shows the SpamAssassin scores obtained by our scoring process using the TR-IRLS algorithm. The “GA generated SA scores” curve was made using the scores generated by the SpamAssassin project using the Genetic Algorithm.

While the ROC Graphs are interesting to look at, they are not very useful for our performance analysis as the underlying assumption is that the threshold can be set to any arbitrary value. This assumption is valid for the “Straight LR” curve, but this is not valid for the SpamAssassin scores, as the whole idea of immutable rules and score ranges (as outlined in Section 2.5) relies on the assumption that the threshold is 5.0. It is for this reason that we have only included the plot for score set 0.
Figure 4.1: ROC graph for score set 0
Figure 4.2: Close up of ROC graph for score set 0
Chapter 5

Discussion

5.1 IRLS-LARS

As mentioned in Chapter 4, no useful results were produced using the IRLS-LARS algorithm; the algorithm often failed to produce a logistic regression parameter from our data. We believe this occurred due to the limitations discussed below.

5.1.1 Limitations of IRLS-LARS

The Matlab implementation of the IRLS-LARS algorithm is extremely slow for large data sets. As a result, data sets containing approximately 20,000 e-mail messages were used, instead of the full data set of over 1,000,000. The memory used by Matlab in loading large data sets exhausts the available system memory which results in excessive swapping. This is quite different from the TR-IRLS algorithm which runs in seconds on the entirety of our available data.

The Matlab implementation of the IRLS-LARS algorithm frequently returned singular matrix and divide by zero warnings as the constraint $C$ was loosened. This may be due to the occurrence of correlated attributes, which causes the weighted least squares subproblems produced by the IRLS algorithm to be ill-conditioned. Our initial analysis showed that accuracy increases as the constraint is loosened, suggesting that it would be better not to use $L_1$ regularized logistic regression.

One reason the IRLS-LARS algorithm may be slow on our data is that it was designed with real-valued data in mind, and therefore does not contain
any of the optimizations for large binary data sets that the TR-IRLS has used.

5.2 TR-IRLS

The TR-IRLS algorithm was specifically designed for use with large binary data sets like ours, and as a result performed very well on the entirety of our e-mail data. Since the algorithm runs in seconds, it was very easy to test various threshold values in order to optimize performance.

5.2.1 Performance Comparison

From the results presented in Chapter 4 we see that the scores generated by TR-IRLS perform better than the scores generated by the GA for score sets 0 and 1; the scores generated by TR-IRLS have a higher TCR. It is interesting to note that this increased TCR is achieved by a much smaller false positive rate at the expense of a larger rate of false negatives. Since TCR is designed to take into account the relative cost of false negatives and false positives, this tradeoff is, by definition, an improvement.

For score sets 2 and 3, TR-IRLS does worse than the GA. This is because the scores we generated using TR-IRLS for score sets 2 and 3 were generated using the parameter weights for score sets 0 and 1 and scaled using a different probability threshold. Score sets 2 and 3 are identical to score sets 0 and 1 (respectively) except that sets 2 and 3 also include the Bayes rules, described in Section 2.4.2. These rules are marked as immutable, so our process, if used directly, would try to determine weights for these rules and then simply discard them. If the Bayes rules were marked as mutable, the TR-IRLS algorithm would still try to assign many of them large (in absolute value) scores that are outside their allowable score ranges, and again the generated scores would be greatly changed.

Instead of generating scores that would not be used, and which would affect all the other rules' scores, we used the scores from score sets 0 and 1, and simply used the immutable scores that already existed.

The GA is able to take into account these immutable rules, and optimize the scores accordingly. If our TR-IRLS algorithm was able to do this, it may perform much better than the GA.
5.2.2 Runtime Comparison

While we did not do an exact runtime analysis, we do know that the TR-IRLS algorithm takes minutes to generate scores, while the GA takes hours. While at the moment, the SpamAssassin project regenerates scores very rarely (at every major release), the project would like to start doing more frequent rule and score updates, so decreased runtime is valuable.
Chapter 6

Conclusion

Based on the results shown in Chapter 4, we believe that logistic regression using the TR-IRLS algorithm shows promise as a score generation mechanism for Apache SpamAssassin. The scores generated by this algorithm are superior to those generated by the currently used algorithm, the Genetic Algorithm (GA), in certain situations.

Our system for score generation is much faster than the GA. We can generate scores in minutes instead of hours. This is a benefit to the SpamAssassin project as it will make it easier to generate scores more frequently.

Work still needs to be done to automate the manual steps of our system; currently, far too much human interaction is required. Also, more work will need to be done in order to improve our design to properly account for score ranges and immutable rules, in order to better compete with the GA. This needs to be done inside the TR-IRLS algorithm, instead of after the scores are generated.

Possibilities for future exploration of our method may also include a comparison between our system and the Perceptron once the problems it is experiencing are worked out. Since the runtime of the Perceptron is very comparable to our system, it would be interesting to see a performance comparison relating the TCR values between the two methods.
Bibliography


