Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Performance Evaluation Of Spam Detection Techniques In Relation To Stream Computing

Author: Hugh Lavery

Supervisor: Stephen Barrett

A Dissertation submitted in partial fulfilment

of the requirements for the degree of

MAI (Computer Engineering)

Submitted to the University of Dublin, Trinity College, April,

2019

Declaration

I, Hugh Lavery, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signed: _____ Date: _____

Summary

This research explores algorithmic adaption as an alternative to load shedding in stream computing. Algorithmic adaption is a concept for altering the algorithm of computation in a stream computing system. Specifically, algorithmic adaption switches out the computational algorithm in a stream computing system under load to one which is less expensive to increase the throughput of messages.

To explore this concept, a number of machine learning algorithms for content based spam filtering were implemented and evaluated with an eye to the performance and cost requirements needed for algorithmic adaption to be feasible.

Spam emails is a continuing issue. A large portion of emails in circulation are spam emails. Spam detection systems must keep the number of false positive predictions to a minimum. False positive prediction corresponding to a real email being filtered out of a users inbox. As such, detection techniques must minimise false positive rate and there must be different costs associated with using different techniques.

This work finds that algorithmic adaption is a feasible solution to load shedding and outlines the next stage in the research on algorithmic adaption.

Abstract

This dissertation explores an alternative to load shedding in a stream computing environment. The hypothesis which leads this research states: "Is there viability to implement adapting/switching out the computational algorithm in a stream computing system under load, where the computations being done are sufficiently complex and alternatives with a trade of in effectiveness and regained cost are available". To explore this, the application of content based spam detection was chosen as it fits a stream computing environment and requires sufficient computations. A number of machine learning models were implemented and evaluated in terms of effectiveness and cost and conclusions were drawn on the effectiveness of algorithmic adaption as an alternative to load shedding.

Acknowledgements

I would like to thank my supervisor Stephen Barrett for all the help and guidance he has given me. He always kept me on track and helped keep the research moving forward. I am incredibly grateful for all the time and effort he has put in to help me and this dissertation would not have been possible without him.I would also like to thanks my family and closest friends for their continued support in whatever I decide to do.

Contents

1 Introduction **1**

1.1 Stream Computing 1
1.2 Spam Detection Techniques 2
1.3 Evaluation 4
1.4 Key Findings 5

2 State of The Art **6**

2.1 Stream Computing 6
2.1.1 Data Based Solutions 7
2.1.2 Task Based Solutions 9
2.1.3 Computational Shedding 9
2.2 Machine Learning 10
2.2.1 Cost/Loss Functions 13
2.2.2 Optimisation Methods 13
2.2.3 Regularisation 16
2.2.4 Logistic Regression Classifier 16
2.2.5 Support Vector Machine (SVM) 19
2.2.6 Softmax Classifier 20
2.2.7 Artificial Neural Networks 22
2.3 Haskell 26
2.3.1 Haskell and Machine Learning 26
2.4 Spam 27
2.4.1 Spambase Data Set 28
2.4.2 Related Work 29

3 Design **33**

3.1 Gathering Training/Testing Data 33
3.2 Pre-Processing Training/Testing Data 34
3.3 Build model 35
3.4 Train model 35

3.5 Evaluate Performance . . . 35
3.5.1 Cross Validation . . . 36

4 Implementation **37**
4.1 Haskell . . . 37
4.2 Machine Learning . . . 37
4.3 Time measurements . . . 38
4.4 Measurements . . . 39
4.4.1 Accuracy . . . 40
4.4.2 False Positive Rate . . . 40
4.4.3 Precision . . . 41
4.4.4 Recall . . . 41
4.4.5 F1 Score . . . 41
4.5 Model Parameters . . . 42

5 Results **43**
5.1 Cross Validation . . . 43
5.2 Time Measurements . . . 43
5.2.1 Training Times . . . 45
5.2.2 Prediction Times . . . 45
5.3 Performance Measurements . . . 46
5.3.1 Accuracy . . . 46
5.3.2 Precision . . . 47
5.3.3 Recall . . . 48
5.3.4 F1 Score . . . 49
5.3.5 False Positive Rate . . . 50

6 Evaluation **52**
6.1 Is Algorithmic Adaption a Feasible Solution? . . . 52
6.2 Comparison to previous work. . . . 53
6.2.1 Performance . . . 53
6.2.2 Cost . . . 54

7 Conclusion and Future Work **56**

List of Figures

2.1 Flow diagram of procedure for training supervised learning algorithms. . 12
2.2 Gradient descent step 1 14
2.3 Gradient descent step 2 15
2.4 Gradient descent step 3 15
2.5 Logistic Regression Probability Plot 18
2.6 2D Support Vector Machine 19
2.7 Kernel Method 21
2.8 Neural Network Structural Representation 24
2.9 SVM Results from (1) 30
2.10 ANN Results from (1) 30
2.11 ANN Results from (2). Columns are Accuracy, Precision, Recall, Accuracy and F1 score from left to right. 31
2.12 Results from (3) 32

4.1 Confusion Matrix 39

5.1 Neural Network 10 Fold Cross Validation Results 44
5.2 Training Times 45
5.3 Prediction Times 46
5.4 Accuracy 47
5.5 Precision 48
5.6 Recall 49
5.7 F1 Score 50
5.8 False Positive Rate 51

List of Tables

1.1 Measurements 5

2.1 Neural Network Activation Functions 25
2.2 Different Types of Spam 28

5.1 Time Measurements 44
5.2 Performance Measurements 47

Nomenclature

SVM	Support Vector Machine
NN	Neural Network
ANN	Artificial Neural Network
SM	Softmax Classifier
GD	Gradient Descent Optimisation
BFGS	Broyden–Fletcher–Goldfarb–Shanno Optimisation
PCA	Principal Component Analysis

1 Introduction

The hypothesis which lead to this research came from the need for an alternative to load shedding in stream computing environments. Some research has been done into computational shedding which is the fine grained removal of computations in an attempt to regain processor cycles. While computational shedding is effective in certain situations there is a need for a more generalised approach in order to increase the throughput of input data when a stream computing system is under load. The proposed solution is called Algorithmic Adaption.

The hypothesis states: "Is there viability to implement adapting/switching out the computational algorithm in a stream computing system under load, where the computations being done are sufficiently complex and alternatives with a trade of in effectiveness and regained cost are available". To explore this hypothesis, implementing a number of spam detection models to evaluate their performance when detecting spam was chosen. Spam email fits a stream environment as mail servers must deal with unpredictable email rates as they arise and the detection techniques are sufficiently complex.

1.1 Stream Computing

Stream computing is a processing model which aims to provide a different approach to processing large volumes of data. The traditional approach is batch processing where the data is pulled by a Database Management System (DBMS). In a stream computing system, data is pushed from one or more data source as the data becomes available. Data sources such as these are real time data streams. Stream computing is particularly useful in situations where the system has real-time constraints, and as such and the output of the system needs to be continually computed.

Real time data streams pose a potential problem for stream computing systems. The

system will have a limited to the rate at which it can process data, typically a combination of processor cycles, memory, and bandwidth. The system must be able to cater for the possibility of the data stream exceeding the rate at which the system can process the data. The typical solution to this is to drop input data at random from the stream, which is known as load shedding. While this may be an acceptable compromise for some systems it is definitely not in others.

This dissertation attempts to explore the viability of an alternative to load shedding to avoid this explicit data loss. The proposed solution was algorithmic adaption. This solution proposes changing the algorithm to one which is less computationally expensive and typically less accurate during periods of increased data rates. The aim is that there will be a number of algorithm switches possible where the decrease in processing power is related to the accuracy loss. This alternative solution could then be compared to a data shedding solution.

The solution of a stream computing model with algorithmic adaption will not always be an applicable solution. For the purpose of this exploration I have chosen to take the problem of spam email detection. While this is somewhat of a "solved" problem and systems can deal with the current loads, it fits the stream computing model and there are a number of algorithms of different complexity. The results of this system are measurable, and comparable and will give valuable insight into the possibilities of algorithmic adaption in stream computing systems.

1.2 Spam Detection Techniques

Spam has been an issue for as long as email has been in widespread use. Spam ranges from annoying unsolicited emails to sophisticated phishing attacks which are ultimately looking to steal data or money from the recipient. Spam can be difficult to spot, particularly for those less experienced with technology. It has been reported that well over half of email in circulation is spam. As spam is so varied and dynamic in its content, rule based filters would not be effective. Machine learning models are able to spot patterns in data and as such are much more effective at filtering out spam.

Spam detection filters use machine learning models to predict if a given email is spam or not. These models must be trained with a labeled data set of spam and ham (non spam) email. Early models used include Bayesian classifiers and Support Vector Machines but more recently Neural Networks have been used due to their increased performance.

The chosen programming language to implement the spam detection models was Haskell. Haskell was chosen as it is a strong candidate for implementing both stream computing systems and machine learning models. The Haskell machine learning community is not as mature as that of Python or R, which poses its own issue but it is certainly worth exploring machine learning and Haskell to see if it can meet these expectations.

The machine learning models looked at were:

1. Support Vector Machine (SVM)
    - Gradient Descent (GD) Optimizer
    - Broyden-Fletcher-Goldfrard-Shanno (BFGS) Optimizer
2. Logistic Regression
    - Normal Logistic Regression
    - Logistic regression with Principal Component Analysis (PCA).
3. Softmax Classifier
    - Gradient Descent Optimizer
    - Broyden-Fletcher-Goldfrard-Shanno (BFGS) Optimizer
4. Neural Network with Broyden-Fletcher-Goldfrard-Shanno (BFGS) Optimizer

Spam filtering was originally focused on content based filtering. Content based filtering can be defined as parsing the raw text of the email and making a feature vector of key features which are indicators of whether or not an email is spam, and using these feature vectors to predict if the email is spam or not. Examples of these features would be frequency of words which are indicative of spam or ham emails. Words such as "free" or "winner" are often seen in spam emails, in contrast ham emails are more likely to contain the recipients name or keywords related to their field of work.

Modern spam detection such as that done by Gmail incorporates a lot more information than just the content of the email. A number of sources are used to build a "spam score". One of these sources is the raw email content, others include if the ip source address of the email is within certain ranges associated with being a source of spam email or if the sender is on the recipients contact list. Due to the difficulty of access to this information it was decided that the research in this dissertation would be

entirely based on content filtering. This would still provide enough complexity to evaluate if algorithmic adaption is a viable alternative to load shedding.

The data set used is called SpamBase and is available on UCI Machine learning repository(4). It is a pre-processed data set which is a matrix of labeled feature vectors, this allowed the focus to be on the machine learning aspect as opposed to natural language processing. It was randomly shuffled once to ensure all models were evaluated on a level playing field. 10 fold cross validation was used when training and evaluating the models.

1.3 Evaluation

The main measurements used to evaluate the models can be seen in Table 1.1. These measurements are standard evaluation metrics for binary classification machine learning models such as those used for spam detection. Much of the research into spam focuses mainly on accuracy. While that is important, the false positive rate has a bigger impact on the effectiveness of the model as this is the number of real emails being detected as spam and filtered out of the recipients inbox. This is a much bigger issue than a false negative; it is generally accepted that one would rather receive a few spam emails then to miss a real email.

Accuracy and false positive rate are good early indicators of the performance of each model. Precision and recall are calculated from a confusion matrix of the number of true positive, true negative, false positive and false negative predictions. Precision, recall and their harmonic mean (F1 score) can be used to accurately compare the performance of the different models.

After training and evaluating the models it was found out that the resulting prediction times (effectively cost in stream computing) was very similar across most of the models. This was not only true for the cost but also in terms of performance (accuracy, false positive rate, and precision). This was quite a surprising result particularly in terms of time complexity. It was expected that some of the models would be similar in terms of performance but not so closely in terms of cost.

Type	Description
Accuracy	Percentage of the number of correct predictions to total predictions.
False Positive Rate	Percentage of the number of false positives to the total number of positives. In this case the number of ham emails being classified as spam over the total number of spam emails.
Recall	Number of real spam emails that were correctly identified as spam emails.
Precision	Proportion of emails identified as spam (positive) that are actually Spam emails.
F1 Score	Harmonic mean of precision and recall. This combination is a good overall indicator of a models performance and is the best measure for comparing different models.
Training time	Total time taken for model to be trained on entire training set including any optimisation of the models.
Prediction time	Time it takes for model to classify an email as spam or ham. This is a key indicator of computational cost.

Table 1.1: Measurements

1.4 Key Findings

There was a slight trend which makes algorithmic adaption plausible, but there is not enough evidence to prove the hypothesis. This lead onto the next question about these performances and if this is the expected performances of these models. While there is a lot of research on the performance of spam detection algorithms there is very little on their run time performances in any programming language let alone Haskell.

The final section of the analysis is an evaluation of the real time performance of the models implemented. There is not much research in terms of practical or theoretical times when it comes to models prediction times however a comparison is made with what was available.

Alongside the analysis of the time performance of the models, the non time related performance measurements mentioned earlier will be compared to other implementations of these models in other programming languages. This combined with the analysis of the time complexities of the models gives a good consensus of the current state of machine learning in Haskell.

2 State of The Art

2.1 Stream Computing

Stream computing has been in steady development over the last 15 years as an alternative to traditional batch processing systems. Early stream computing concepts were concerned with real time monitoring systems. Early examples would be Aurora and Borealis stream processing engines (5) (6). Aurora was a first generation stream processing system. Borealis is a second generation stream processing system following from the work done on Aurora. Borealis is a distributed system where processing takes part on multiple physical distributed nodes. These early versions focused on defining a new processing architecture for real time systems such as hospital monitoring or financial analysis applications. These early systems used load shedding when the rate of data being pushed by the stream exceeded the systems capabilities, and accept that this may result in approximate answers.

The implementation and drawbacks of load shedding was explored in "Load shedding in a data stream manager" (7). Two load shedding techniques were explored: randomly dropping a fraction of input data (tuples) and dropping tuples based on their perceived importance. The issue explored in (7) is when the system with a number of push based data sources cannot satisfy the rate of incoming data as the arrival rates can be high and unpredictable. The systems which that paper was based around were all monitoring systems looking to detect critical situations, during which data rates can exceed system capabilities. If this is not corrected, latency will increase due to the queues being formed. For these systems the solution is to drop load; it is an attempt to gracefully degrade the performance of the system while concentrating on key quality-of-service information when making load shedding decisions.

While "Load shedding in a data stream manager", makes a compelling argument for the utility of load shedding in monitoring applications, it does not explore stream computing systems which must process each piece of data. Regarding spam for

example, all emails are independent of each other and owing to the large percentage of emails being spam, some emails will be "let through" unprocessed. Similarly in social media content filtering posts must be checked to monitor hate speech among other things. These classification problems require some processing and cannot accept load shedding.

Unlike monitoring systems which have structured data from different streams, classification systems deal with data of the same form and as such cannot use the structure of source of the data as extra information. Due to this, later research considered different approaches to handling bursty data stream events. These solutions can be broken into two areas: data based solutions and task based solutions.

2.1.1 Data Based Solutions

Random Load Shedding

Data based solutions, aim to reduce the processing by reducing the number of data units to be processed. These are typically some variation on load shedding, be it statistically random or informed decisions. As such, a subsection of the input data is not processed. These are explored in “Load Shedding on data streams” (8)

In load shedding solutions, the shedding needs to take place where it has the biggest impact in reducing processing power. This reduction in processing power must be balanced with minimising the number of data elements being dropped. This is different than load shedding in systems such as computer networks, where shedding takes place at any point where a bottleneck occurs across the network, and little concern is placed on which packets are dropped as they can be resent. This would not be efficient in stream processing as the amount of data being discarded is looking to be minimised for a given return of processing power.

Early distributed stream computing systems used directed-acyclic-graphs (DAG), which outline the flow of data through the stream processing node, in an attempt to determine where to place a load shedder to have the biggest effect on reducing processing requirements.

This was efficient for monitoring applications, which had an increase in accuracy when data elements were partially processed before being discarded. In classification based stream computing systems, if an element (email, social media post) was partially processed before being dropped, this would result in wasted processing power with no

added classifications being made.

The exploration of load shedding and its consequences was lead by the research done during the development of the Aurora and later the Borealis stream processing engines. Because of the simple nature of the messages being processed by Aurora (temperature measurements) and the simple operations performed, it was easy to accurately measure the required process cycles for each message and as such, was easy to determine when the system is in or approaching an overloaded state. This allowed them to dynamically drop messages at processing engines which were overloaded until the system could deal with the load.

Aurora used random load shedding. It treated all input messages with equal importance and dropped messages at random. Borealis dropped messages in groups depending on how overloaded the system was and messages were all treated with equal importance.

In the case of Aurora, they describe using a greedy algorithm to perform load shedding. This algorithm identifies types of messages whose output has the smallest negative slope to the QoS (quality of service) graph. They then moved along the graph until they found the second output which had the next lowest negative slope. From this it chooses which output tuples to drop, resulting in the minimum decrease in overall QoS. This solution can be done statically before the system is live or dynamically during run time. The run time case looks to drop problematic outputs based on the QoS graph and drops them until the system is no longer overloaded.

Semantic Load Shedding

The random load shedding solutions mentioned treat all messages equally and take no regard for the impact of dropping certain messages on overall system performance. Semantic load shedding looks to drop less important tuples.

Semantic load shedding is investigated in "QoS-Driven Load Shedding on Data Streams" (9). These techniques look to control the performance degradation when the system is under load by dropping input which is less relevant. They also look at assigning Quality of Service (QoS) metrics to input messages so that when the system is overloaded it can drop ones which have less of an effect on the QoS of the system.

In terms of spam detection each email is as relevant as the next, as such no QoS related information could be attached to the message. This is perhaps a possible

approach in the case of mail server providers, but not in content based filtering.

2.1.2 Task Based Solutions

To other solution to an overloaded stream computing system are task based approaches. There are two approaches of note in distribute stream computing system. The first being a form of load balancing where the system transfers tasks from overloaded nodes to ones which are free to process or less overloaded. The second is scaling the system, adding more nodes until the system can handle the current increased load.

These solutions are explored in "Stormy: An Elastic and Highly Available Streaming Service in the Cloud"(10). The load distribution approach requires a distributed hash table to manage sharing of tasks when a node is overloaded. This is a decentralised system and there is some overhead in this management meaning there is a delay in the system responding to the overload. Stormy also explores the solution of "Cloud bursting" which is bringing new nodes online when the system as a whole is overloaded. This is a quite effective approach when the overload is long lasting but is not effective for short lived overloads due to the time taken to bring a new node online and routing tasks to it.

2.1.3 Computational Shedding

Guerin's PhD "Computational Shedding in Stream Computing" (11) explores stream computing models which have to deal with infrequent but intense bursty events (i.e short lived increase in input rate above the systems capabilities). In this context the task and data based solutions are not acceptable. The event is too short lived for the tasked based solutions to be viable and the system cannot accept data loss through load shedding. Guerin proposes computational shedding by temporarily disabling sub tasks, so each input message/event has lower processing cost. This is effective when the computations has sub tasks which can be excluded without a large sacrifice to the accuracy of the system.

While Guerin's solution is effective in some solutions it raises the question is there a more generalised approach to reducing the computational cost of a message/event through substituting the computational algorithm being used. As with Guerin's work, there would be some sacrifice in accuracy by switching to a less accurate model. If the sacrifice in accuracy corresponds to reduced processing cost of a message and is less

than the accuracy loss due to load shedding, then it could be a viable solution to load shedding.

2.2 Machine Learning

Machine learning is a type of Artificial Intelligence that has been in development since the late 1950s. Machine learning is based on statistical models which have been present for hundreds of years. Machine learning techniques work by finding patterns in data and making inferences from these. There are three categories of machine learning;

- Reinforcement Learning
- Supervised Learning
- Unsupervised Learning

Machine learning has seen a major resurgence since the early 2010s when neural networks started to out perform other machine learning algorithms and computation power was becoming cheaper and more widely available. The first occurrence of neural networks drastically outperforming other models was when Alex Krizhevsky created a convolutional neural net which won the ImageNet challenge in 2012 (12). This caused a sharp increase in the research being done on Neural Networks and since then they have been shown to be able solve more complex problems that traditional machine learning models could not.

Reinforcement Learning

Reinforcement learning models complete a goal or task (action) in a live environment, and based of the results of this action they are fed in a reward in a feedback loop(13). Based on how good or bad the result and the magnitude of it, the model iteratively learns if it is making good actions or bad. Unlike the other categories of models, reinforcement learning models continue to learn as they are deployed. Reinforcement learning is good for applications where the problem being solved is likely to change over time and is non static. Where other models are trained offline, reinforcement algorithms learn live as they are deployed.

Reinforcement learning is most commonly used in gaming applications. It also has uses

in robotics where the robot is interacting with the environment around it or personalised recommendation systems where the model learns if users choose its recommendations. Reinforcement learning is not an ideal for natural language processing tasks such as spam detection so there are no reinforcement learning models investigated in this dissertation.

Unsupervised Learning

Unsupervised learning is the branch of machine learning which is concerned with labelling or classifying unlabelled data. These types of models do not learn from feedback such as labels in training data or responses from an environment. Unsupervised learning relies on identifying commonalities between data points and clustering the data points based on these commonalities. It has some use cases in statistics and customer segmentation (based off buying history/key words in social media posts).

As spam detection can be done from labelled data sets, unsupervised learning would not be an optimal choice for spam detection so no unsupervised learning techniques are investigated in this dissertation.

Supervised Learning

The final category of machine learning is supervised learning. It is by far the category with the most research and has the largest number of different models available. It relies on having a label associated with each feature vector and is used for both classification and regression applications.

As this dissertation focuses on classifying spam email I will be focusing on supervised learning techniques for classification. Supervised learning models build an inferred function form the set of data vectors and accompanying labels. This function can then be used to predict classes of unseen data vectors without an accompanying label/class.

The typical approach to training supervised learning algorithms is to get a set of labelled training data. This data is then split into a training set and testing set. A common ratio to split the training data is 70% training and 30% testing data. Once the model has been trained with the training set, the performance of it is evaluated based on the model's predictions for the testing set versus the real values.

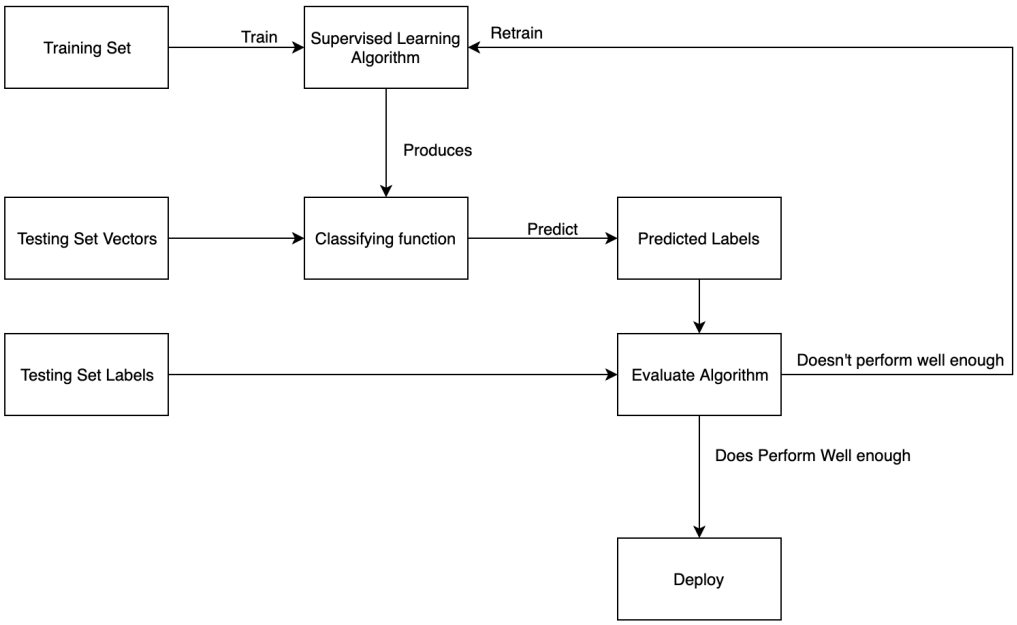

Figure 2.1: Flow diagram of procedure for training supervised learning algorithms.

Different models have different optimisation algorithms and loss function, and in the case of neural networks activation functions. I will investigate these in the following subsections where I describe the supervised learning algorithms investigated for this dissertation. These optimisation algorithms, activation functions, and loss functions have parameters which need to be tuned for each specific use case of the models. Figure 2.1 shows the typical development flow of supervised learning algorithms. I will explore how to evaluate machine learning models in the section 4.4 on measurements.

Supervised machine learning models can be broken into two categories; probabilistic and deterministic. Probabilistic models build a probability distribution over the training set of data for each class and use this to predict the class of unseen data. Deterministic models separate the vectors from the training set in the feature space and associates these spaces with a class. They do this by defining a hyper-plane. New feature vectors can then be classified from this.

In the following subsections I will first outline the cost functions, optimisation methods, and regularisation which are common across all the models. I will then outline the following algorithms which were investigated for this dissertation, including any other techniques which are unique for each of these models.

1. Logistic Regression

2. Support Vector Machine (SVM)

3. Softmax Classifier

4. Neural Network

2.2.1 Cost/Loss Functions

Each model has cost/loss function. The loss function takes the prediction for a given input and calculates the error in relation to the real value for the input (during model training). The size of the error term is then used to teach the model how wrong or right its prediction was. Optimisation methods use the result of the loss function during training to tune the parameters of the model in such a way to minimize the loss.

Examples of loss function include cross entropy loss (used in softmax classifier) and hinge loss (used in SVM). Other examples include mean square error loss, negative log likelihood, logistic, and cosine proximity.

2.2.2 Optimisation Methods

The following optimization techniques can be used during the training stage of all machine learning models. There are a number of optimization techniques but they all look to minimize a loss function to its minimum by changing the models parameters during the model training.

Gradient Descent

Gradient descent looks to minimise the loss function of a given model to its local minimum. Gradient descent and its variations are described in (14). Gradient descent starts with initial parameters for the loss function, or in the case of neural networks random weights for each node. It then computes the gradient at that point and takes a step in the direction opposite the ascending gradient. It iterates until it hits the minimum of the loss function. Due to the possible presence of a number of local minimums it is not guaranteed to converge on the global minimum. See figure 2.2 - 2.4 for an example of gradient descent steps converging on a local minimum.

The size of the step taken in gradient descent is decided by the learning rate. The smaller the learning rate the smaller the step take. If the learning rate is too big it is

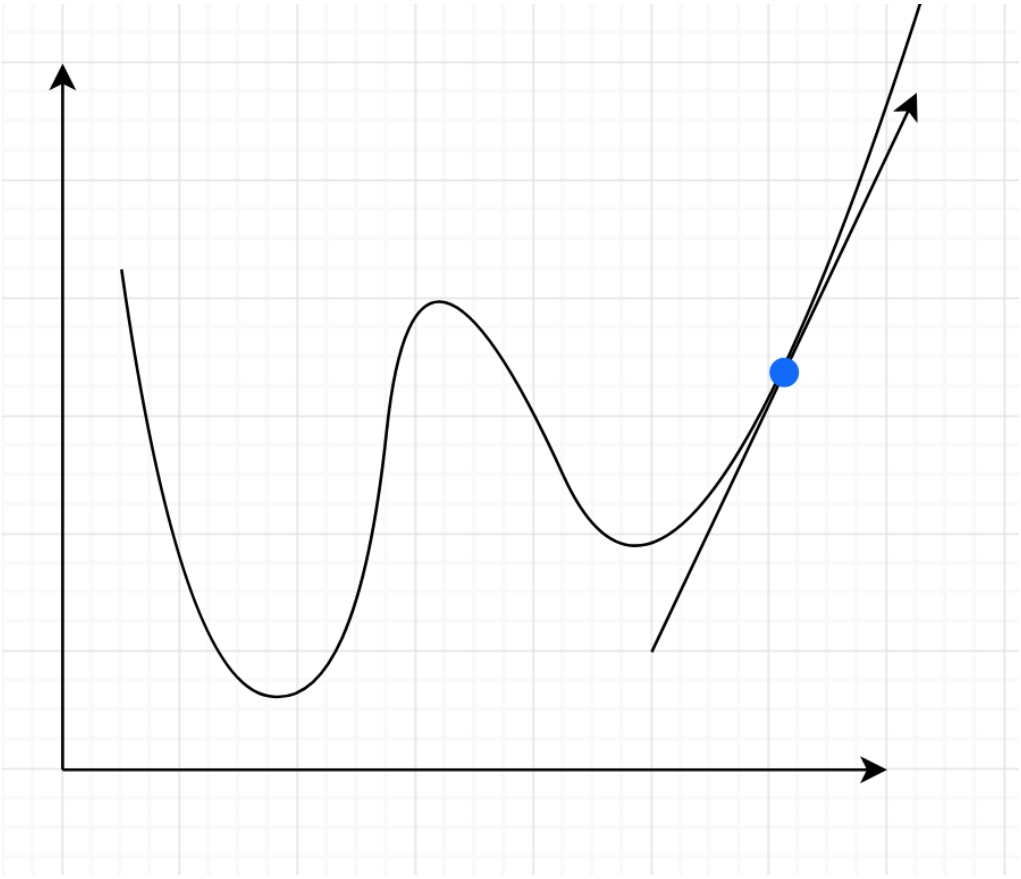

Figure 2.2: Gradient descent step 1

possible to overshoot the local minimum. With a very low learning rate you are guaranteed to converge on the closest local minimum to the starting point but it could take a very long time to converge. Typically very small values such as 0.001 are used.

Broyden–Fletcher–Goldfarb–Shanno (BFGS)

The other optimization method looked at was the BFGS method. As with gradient descent, BFGS looks to find the point where the gradient is zero. This optimisation technique falls into the family of hill climbing optimization techniques, specifically a quasi-Newton method(15). A quasi-Newton method is one which is similar to Newton's method for finding local minimum/maximums but without having to compute the Hessian (matrix of second order partial derivatives). Instead the Hessian is approximated from the gradient.

As with gradient descent, BFGS looks to find the minimum of the loss function. Gradient descent relies on computing the first derivative of the loss function whereas BFGS relies on the second derivative. After each iteration of training, gradient descent

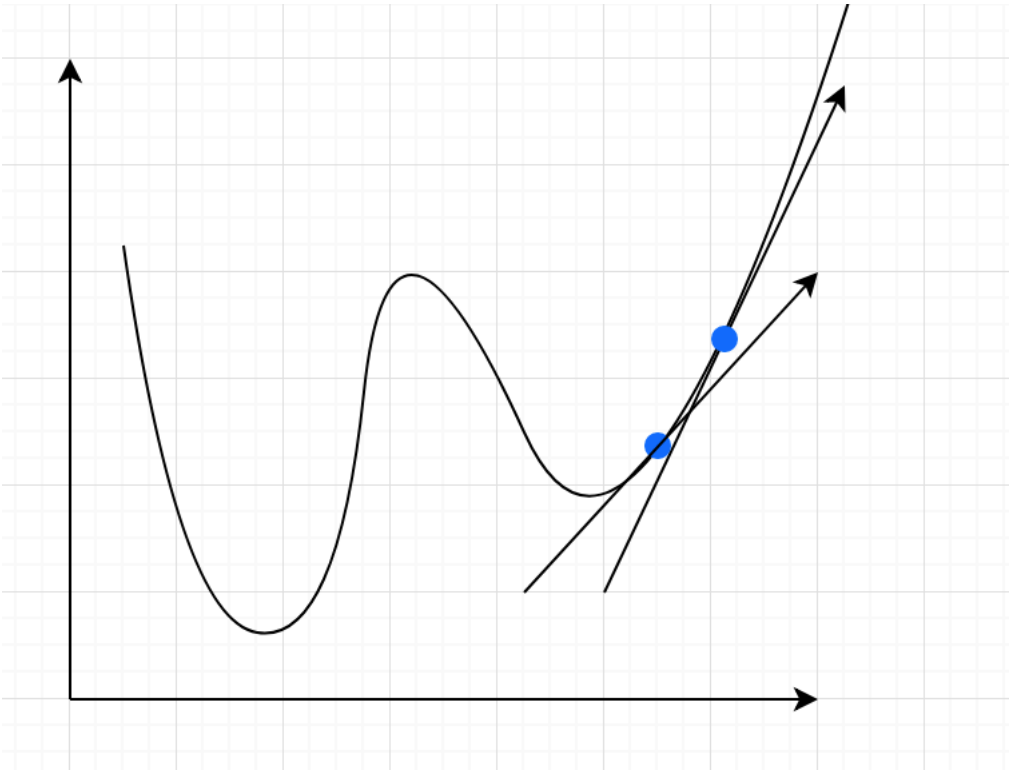

Figure 2.3: Gradient descent step 2

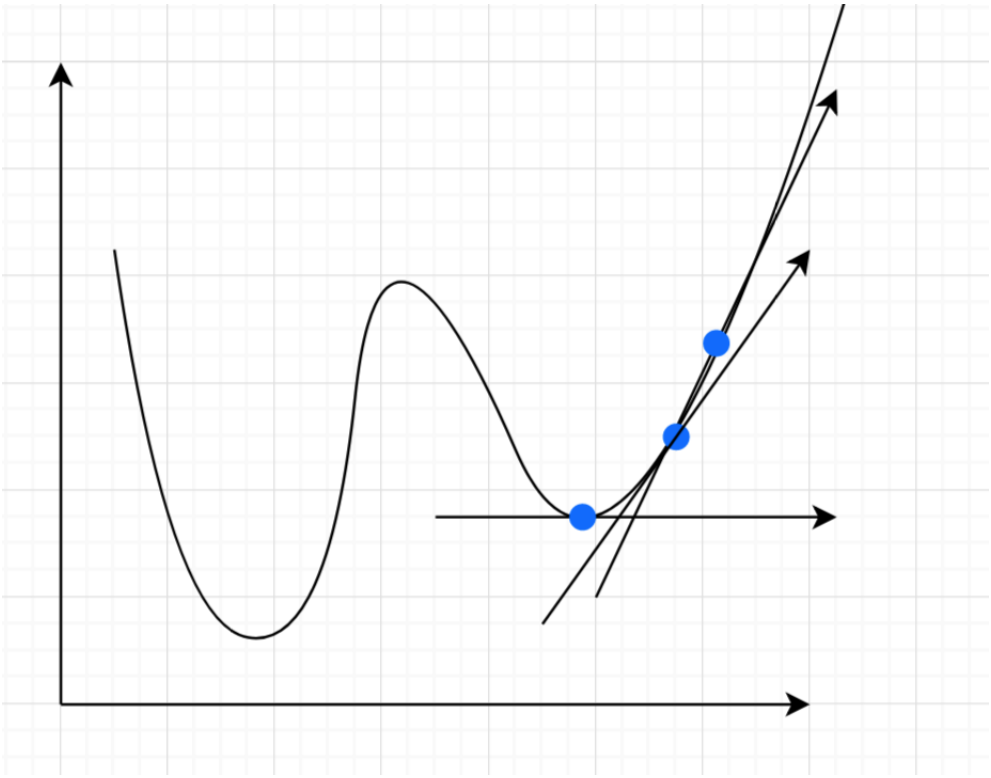

Figure 2.4: Gradient descent step 3

will adjust all the parameters in the loss function where hill climbing techniques such as BFGS adjusts one parameter at a time.

BFGS performs well when the feature space is relatively flat but is more expensive in terms of time and memory.

2.2.3 Regularisation

Regularisation helps to avoid overfitting the model to the training data. This term helps reduce the variance of the model by simplifying it such that is does not represent the training data too closely. This is done to help combat noise in the training data.

The most common regularisation terms are L1 and L2 regularisation. L1 terms adds a penalty which is equivalent to the value of the magnitude of the coefficients. L2 regularisation is equal to the square of the magnitude of the coefficients.

L1 Regularisation:

$$\lambda \|\vec{w}\|$$

L2 Regularisation:

$$\lambda \|\vec{w}\|^2$$

L2 term has been shown to be more effective at reducing overfitting. As described by Yichuan Tang(16) in his paper on deep learning using SVMs "L2-SVM is differentiable and imposes a bigger(quadratic vs. linear) loss for points which violate the margin" resulting in reducing the effect of noise in the training set. As such, L2 regularisation is used in all the models evaluated.

2.2.4 Logistic Regression Classifier

Logistic regression is the oldest machine learning classifier and was discovered long before the term "Machine Learning" was coined. The original discovery of logistic regression is attributed to Pierre-François Verhulst in 1845. It wasn't until 1920 when it was rediscovered independently by Raymond Pearl and Lowell Reed that it began to gain momentum in the statistics community(17).

The form of logistic regression used in spam detection is a binomial regression. There

are two possible outcomes; 1 - email is spam, or 0 - email is not spam. However there are extensions to logistic regress for the case where there is multiple classes in the data-set. Logistic regression is a probabilistic model. It is an extension to linear regression which is used for predicting continuous values (not classes).

Logistic function:

$$Logistic(n) = \frac{1}{1 + \exp^{(-n)}}$$

Where n is:

$$n = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_f x_f$$

and f is the number of features used to represent the underlying thing being classified. And

$$\beta_i$$

are the parameters from the trained model.

Logistic regression predicts the probability that a given email is spam. The resulting probability curve can be seen in Figure 2.5, in this example with only 1 feature.

In addition to a normal logistic regression classifier being implemented one was implemented after Principal Component Analysis (PCA) was performed on the data set prior to the model being trained.

Principal Component Analysis (PCA)

Principal component analysis is a technique to reduce the dimensions of the features used when a model is trained and when it predicts a value. It deduces which features are contributing to the variance in the model. PCA looks to maximise this variance. It reduces the input features into a set of principal components which help improve the performance of the model by effectively removing some features which are noisy and do not positively contribute to predictions.

It is important that this same selection of principal components is done to feature vectors that are used outside of training the model. This must be done so that the same principal components are used when predicting the class of unseen data. This increases the performance of the model it will increases the training and prediction times as extra pre-processing must be done.

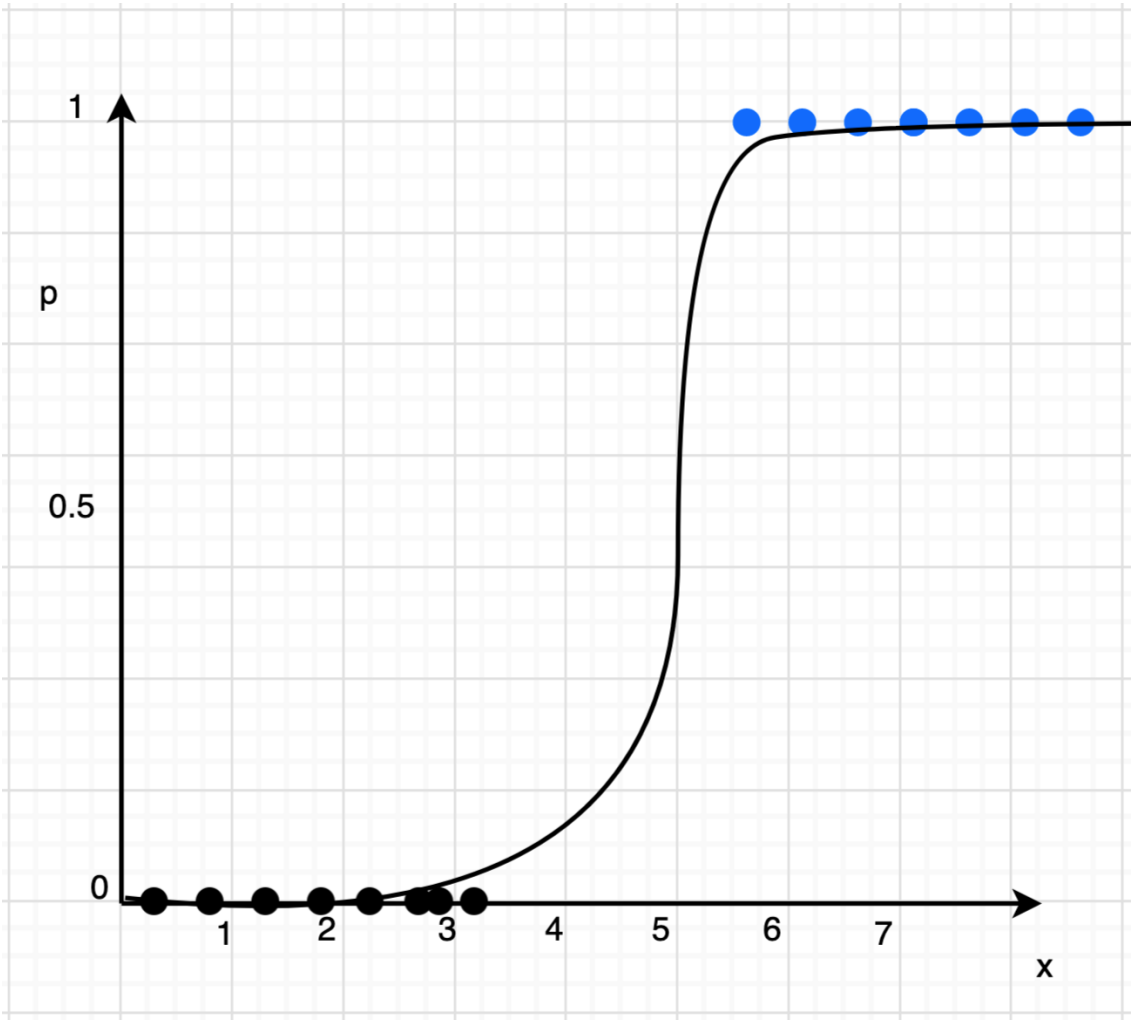

Figure 2.5: Logistic Regression Probability Plot

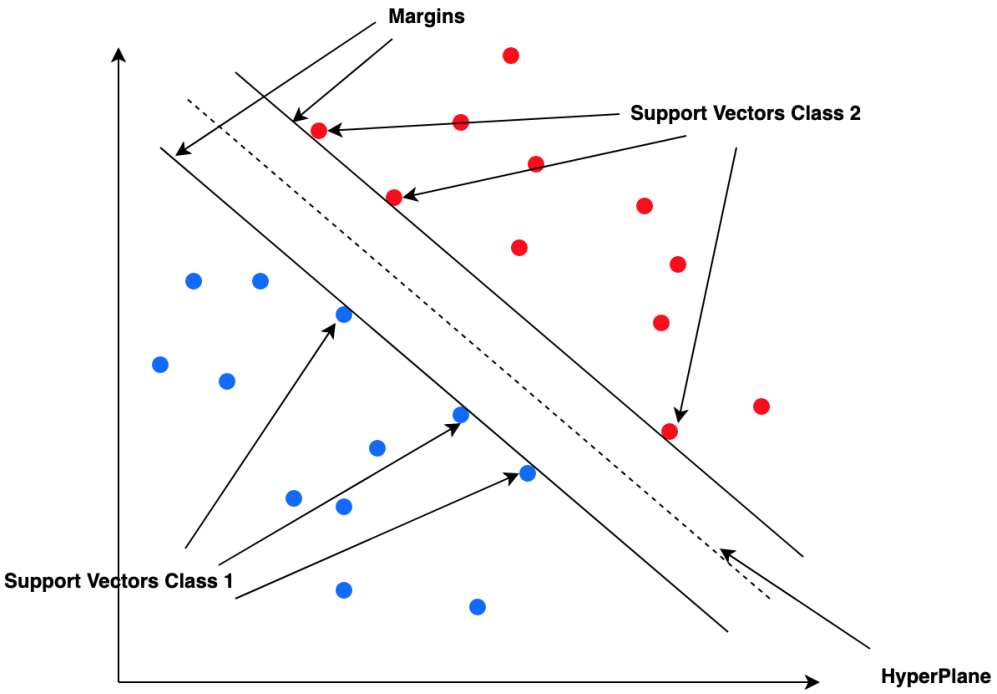

Figure 2.6: 2D Support Vector Machine

2.2.5 Support Vector Machine (SVM)

Support Vector Machines can be used for both linear and non-linear classification(18). This is a deterministic model. Consider the simplified example of an SVM in Figure 2.6. The data points are 2 dimensional and there is two classes of data. The support vectors are the data points which define the width of the margin for the hyper-plane. Unseen data-points are classified by where they plot on the feature space (i.e which side of the hyper-plane they are on). The hyper-plane is the set of points satisfying

$$\vec{w} \cdot \vec{x} - b = 0$$

and

$$\vec{w}$$

is the normal vector to the hyper plane.

The Support Vector Machine separates the data while maintaining a the largest margin as possible. This is an example of classifying linear data. If the data is non linear then a kernel method is used to transform the data into a different dimensional

space where there is a clear hyper-plane between the data points.

If the data is linear separable then there is said to be a hard margin and the aim is to define the hyper-plane such that there is a maximum distance between the support vectors of each class and its hyper-plane. The function to minimize in this situation is:

$$y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1 \tag{1}$$

Where b is the current prediction and the actual target/label/class for vector i is

$$y_i$$

If the data is not linearly separable then the hinge loss function is used. Hinge loss is typically still used when the data is linearly separable as it has increased performance.

$$\left[\frac{1}{n}\sum_{i=1}^{n}\max\left(0, 1 - y_i(\vec{w} \cdot \vec{x}_i - b)\right)\right] + \lambda\|\vec{w}\|^2$$

The second term is the L2 regularisation term.

Kernel Functions maps data points from one dimensional space to another. These are necessary when the data is not linearly separable. An SVM can only separate linear data so a kernel function must be used to make the data into a feature space where it is linearly separable. Examples of more popular kernel function would include Gaussian radical basis function (RBD) and sigmoid kernel functions. See figure 2.7 for an example of a kernel method in a 2 dimensional space. As the data set used in this dissertation is linearly separable no kernel functions are used.

2.2.6 Softmax Classifier

The softmax classifier is very similar to the SVM in structure. The hinge loss function in SVM gives you the margin separating the two classes of data, where as the loss function for softmax is cross entropy loss. The cross entropy loss function is:

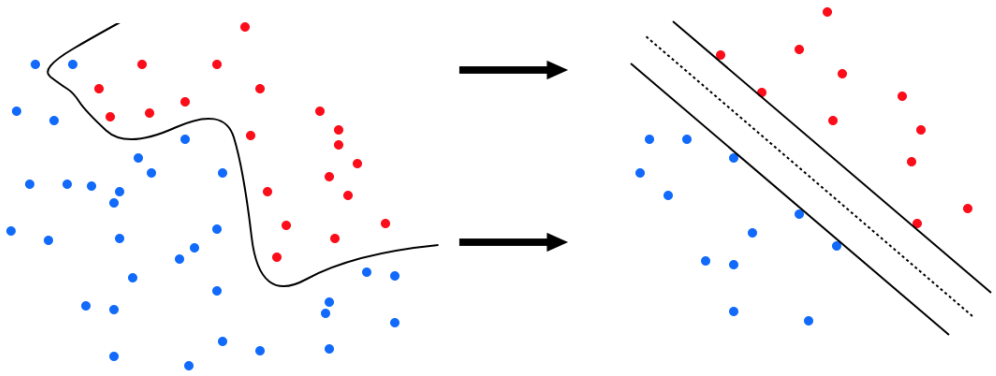

Figure 2.7: Kernel Method

$$L_{yi} = -f_{yi} + log \sum_{j} e^{f_j}$$

where fj is the jth element of vector and the softmax function is:

$$j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Finally, the cross entropy between the estimated distribution q and true distribution p is as follows:

$$H(p, q) = -\sum_{x} p(x) log q(x)$$

This allows the classifier to minimize the cross entropy between the true distribution and estimated class probabilities.

The performance of the softmax classifier and SVM are very similar. However, due to the fact that the softmax computes the probability that a feature vector belongs to a class its output also allows us to determine the confidence it has in the prediction. If both and SVM and Softmax classifier predicted the probability that two vectors are in a class the results would be something like:

$$Softmax = [0.89, 0.11]$$

$$SVM = [43.52, -19.63]$$

2.2.7 Artificial Neural Networks

Artificial Neural networks (or just Neural Networks) were first introduced in 1943 when Warren McCulloch and Walter Pitts came up with a computational model for neural networks(19). The model is based off how biological neural networks in the brain function.

An artificial neural network lays out the structure of how to build a model but uses other machine learning algorithms internally. As such, it is more of a framework for machine learning solutions as opposed to being an algorithm itself. Each node in a neural network represents a neuron in the brain. Each of these neurons takes a weighted sum of its inputs, adds some bias and passes these through its activation function to decide if that particular neuron "fires" (brain related term applied to machine learning). A neuron firing or not determines the inputs to the next layer of neurons. The weights and biases get configured during the optimization phase of the models training.

Research into artificial neural networks stagnated in 1969 when Marvin Minsky and Seymour Papert discovered two major issues with computations and artificial neural networks(20). The first issue was that computers at the time just did not have the computational power required to solve large neural networks (networks with many layers and/or large number of nodes), due to this networks at the time were only single layered networks called perceptrons. The other issue was that these perceptrons could not process the exclusive-or function, which boiled down to being a constraint of perceptrons. This lead to the introduction of multi-layered networks, but due to processing constraints it was a number of years before these could be explored further.

Research into neural networks advanced again in 1974 when Paul J. Werbos released his thesis which introduced the backpropagation algorithm(21). The backpropagation algorithms is key to computing gradient of the loss function in neural networks.

Backpropagation

Backpropagation is a technique which takes the error computed at the output of the model and propagates this back up through the layers of the models and adjust the weights at each node. This makes training a neural network a two stage training. Firstly the training vectors are forward propagated through the nodes and the output is used to predict the class of that vector. The error on this is calculated against the true

value of the class. This error is then propagated to update the weights accordingly.

Backpropagation uses the chain rule to calculate the gradient for each layer in the network. This gradient can then be used by an optimization technique such as gradient descent to update the weights of each node in the layer. Backpropagation helps the network to learn the internal structure (weights) to learn a mapping of the feature inputs to output classes.

Neural Network Structure

A neural network must have at least two layers. These are the input and output layers. The input layer contains the same number of neurons/nodes as there are features in the feature vector. In the case of the spambase (see 2.4.1) data set 57 nodes with an optional 1 bias node. The output layer consist of 2 node which output the predicted probability of the feature vector being spam and ham respectively.

The other layers are called the hidden layers. There can be any number of hidden layers and each of these layers can have a different number of nodes. There is a lot of debate over the number of hidden layers to use, with arguments that two layer networks are sufficient for most problems but other examples show networks with 120 or even 1000 layers being most effective for image classification such as those used in resNet(22). The width of the layers (number of nodes) is also debated but the general consensus is that keeping the layers narrower is better. Narrower layers have quicker training times and their performance is comparable to much wider ones.

See Figure 2.8 for a representation of an example binary classification neural network with 4 input features, 2 hidden layers with 8 and 5 nodes, and an output layer with 1 node.

Activation Functions

The activation function is applied to each node in the network. The activation function decides the output of a given node based on its inputs. The inputs to a node have their nodes weights applied, these inputs are summed up and applied to the activation function and the result is the output of that node which in turn acts as inputs to the nodes in the next layer.

When neural networks were originally created the heaviside function was used as it

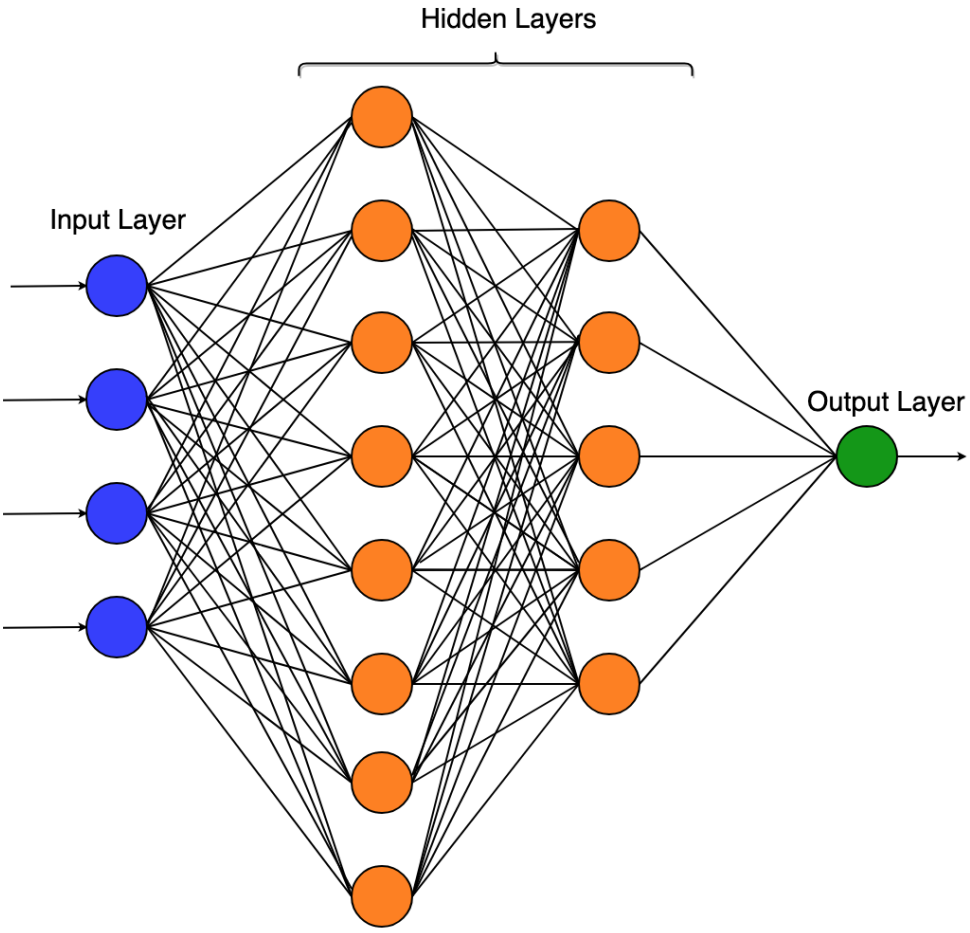

Figure 2.8: Neural Network Structural Representation

Function	Formula
Heaviside	$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
TanH	$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$
Sigmoid	$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$
Rectified Linear Unit (ReLU)	$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
Leaky rectified linear unit (Leaky ReLU)	$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

Table 2.1: Neural Network Activation Functions

approximated the function of neurons in the brain. The heaviside function is the unit function where the output is zero for negative inputs, and one for positive inputs. This was to represent a neuron in the brain firing or not. This worked fine for linear classification problems but is not as effective for non-linear problems.

The other activation functions output a range of values typically between either -1,1 or 0,1 but these can be normalised into the required range.

Some commonly used activation functions can be seen in table 2.1. As mentioned x is the sum of the inputs with their weights applied. Normalisation is not necessary between layers but can help performance.

This dissertation covers artificial neural networks but does not consider convolutional neural networks (CNN) and recurrent neural networks (RNN). CNNs and RNNs have been shown to perform extremely well on more complicated tasks such image recognition but are unnecessarily complex for NLP tasks such as content based spam filtering.

2.3 Haskell

2.3.1 Haskell and Machine Learning

Haskell is a purely functional programming language. It has a strong type system. Due to this strong type system and Haskell's compiler, a lot of bugs are caught at compile time that would only be caught at run-time in other languages.

Python and R would be the standard languages for machine learning. They benefit from having a big machine learning community with a lot of well supported libraries and tutorials. This makes it very easy for those from a data science background to be able to quickly build machine learning models.

A lot of Machine learning is concerned with transformations over unchanging (immutable) data points. Performing transformations like these is very easily expressed in Haskell and the mathematical operations are written in pure functions. Pure in this context means that the function cannot have any side effects. In other works, when a pure function is evaluated the state of the program remains unchanged and the same input is guaranteed to provide the same output.

Haskell also supports concurrency out of the box, making it easy to make the kind of scalable solutions. Scalable solutions are particularly necessary in more complicated machine learning application such as convolutional neural networks and recurrent neural networks.

Another feature of Haskell which makes it a strong candidate for machine learning applications is lazy evaluation. Laziness ensures that only what is needed at a given point in time is evaluated. This leads to efficient data pipelines and avoids doing unnecessary work.

While there are some benefits to doing machine learning in Haskell the community is quite young. Because of this there is not the same support for libraries as the other languages mentioned. At the time of writing this there are a few machine open source learning libraries which implement some things well but they are all in a relatively incomplete stage and in the best scenario only implement a subset of loss, optimisation, and activation functions. Due to the current status of supporting libraries, if one was familiar with machine learning but not experienced with Haskell it would not be encouraging to use Haskell. There is a small dedicated community under the name of "Data Haskell"(23) who are working towards building the supports

necessary for Haskell to be used in data science.

2.4 Spam

Spam email is said to be well over 50% of the emails in circulation. Spam encompasses any unwanted email and ranges from merely annoying unwanted advertising to quite dangerous attacks. While some spam emails are quite common and easily recognisable, others can be almost impossible to tell apart from genuine emails (when the spam is pretending to come from a known source to the recipient). Table 2.2 outlines some of the more common types of spam seen today.

One particular example of a con artist scam has been coined “grandparent scam" as typically it is an older person who falls for it. The “grandparent scam" is some variation of the hacked user being on holidays and being mugged or in hospital and needing money urgently. It is typically the more venerable/less tech savvy who fall prey to these types of spam.

One way or another Spam is looking to trick people out of personal details/money or install malware (through attachments) onto their computer. Spam is constantly evolving and needs patter finding machine learning algorithms to filter spam from real email.

Type	Description
Unsolicited Spam Emails	These emails can either come from a source that is known to the recipient, or unknown but are generally just unwanted advertisement and are not of a dangerous nature.
Spear Phishing Emails	These emails attempt to pretend to come from an official, recognised company and often one the recipient uses (common examples include Paypal or other banks). It wants the user to click on a link to an imitation website to enter personal details such as login information or bank details.
General Phishing Emails	This spam comes in many shapes and sizes. One way or another it wants the user to click on a link to download something malicious or enter personal details. Examples would be emails offering to sell adult content or weight loss pills/Viagra.
Con Artist Scam	These emails can come from a known contact (individual as opposed to a company) who has been hacked or unknown contact. They are trying to trick the recipient out of money by impersonating their contact or outright claiming to have sensitive information which will be distributed unless they are paid out. One way or another they want the recipient to send money through some anonymous method.
Failed delivery Spam /Bounce back Spam	This type of spam pretends to be a failed delivery notification of an email sent from the users mailbox. It hopes to confuse the user to open the attached file. This along with other types of spam have executable attachments which install some type of malware when opened.

Table 2.2: Different Types of Spam

2.4.1 Spambase Data Set

The Spambase data set was chosen as the data set to evaluate the models. It is available on the UCI machine learning repository (4), the data set was originally donated by George Forman of Hewlett Packard Labs in 1999 and has been used in

various papers in relation to spam detection.

The data set consists of 4601 instances (emails) which have been pre-processed into feature vectors of 58 features. This corresponds to 57 attributes of an email and one class label to identify if the email is spam (1) or ham (0).

The 57 features correspond to the following information for a given email:

- 48 features correspond to word frequencies. These are values in the range of 0-100 and correspond to percentage of words in the email that correspond to a given word (i.e 1 of these attributes is percentage occurrence of "telnet" in the email).
- 6 features correspond to character frequencies. This is also in the range 0-100 and corresponds to percentage of characters that match a given character (i.e 1 of these is percentage occurrence of "#" in the email).
- 1 feature corresponds to average length of uninterrupted sequence of capital letters.
- 1 feature corresponds to longest sequence of uninterrupted capital letters.
- 1 feature corresponds to total number of capital letter in the email.

2.4.2 Related Work

The spambase data set is used in a number of papers on spam filtering with machine learning. Salwa Adriana Saab's, Nicholas Mitri's and Mariette Awad's paper titled "Ham or spam? A comparative study for some content-based classification algorithms for email filtering"(1) compares the performance of two SVM models to an Artificial Neural network model.

The results in figure 2.9 show the best performing SVM marked in bold. It is worth noting that the c parameter is related to the regularisation term used. This shows the SVM performing very well in both accuracy and precision.

The results in figure 2.10 show the results for their ANN implementation. This shows a neural network outperforming the SVM implementation in terms of accuracy but not precision. Notably though they referenced another paper which reports an ANN with precision of 94.332% which exceeds all precision results in (1).

Acc. (%)	Precision (%)	Recall (%)	Training Time (sec)	Testing Time (sec)	C
91.35	92.75	84.67	1.42	0.1204	1
92.37	93.31	86.87	1.26	0.1037	2
92.96	93.16	88.64	1.13	0.0862	5
93.35	93.33	89.52	1.07	0.0781	10
93.57	**93.42**	**90.02**	**1.09**	**0.0717**	**20**
93.65	93.23	90.46	1.099	0.0663	40
93.74	93.20	90.73	1.23	0.0631	80
93.61	93.13	90.46	1.35	0.0626	100
93.85	93.51	90.68	1.38	0.0616	120
93.87	93.32	90.95	1.48	0.0605	150
93.94	93.28	91.17	1.598	0.0594	200
93.87	93.32	90.95	1.79	0.0595	250
93.85	93.42	90.79	2.10	0.0581	300

Figure 2.9: SVM Results from (1)

Acc. (%)	Precision (%)	Recall (%)	Time (sec)	h
93.50	91.50	92.05	8.25	1
93.52	92.43	91.00	23.64	2
94.02	**92.96**	**91.78**	**70.81**	**5**
93.78	92.82	91.28	66.65	10
93.74	92.81	91.17	250.34	20
93.28	**94.332**	**94.5**	**As reported in [12]**	

Figure 2.10: ANN Results from (1)

MULTILAYER PERSAPTRON	93.28	94.332	94.5	93.28	94.45

Figure 2.11: ANN Results from (2). Columns are Accuracy, Precision, Recall, Accuracy and F1 score from left to right.

The paper cited in (1) is "Adaptive Approach for Spam Detection" by Sharma and Arora (2). Figure 2.11 shows the result for their Multi-Layer Perceptron (alternative name for ANN). This shows an ANN outperforming the SVM in regards to the key evaluation metric (precision). This gives reason to believe that an ANN can outperform the more traditional techniques.

Another related work to be mentioned is Idris paper "Spam Classification With Artificial Neural Network and Negative Selection Algorithm" this paper looks at a particular optimisation technique called "negative selection" (24). While Idris doesn't specifically cite the SpamBase data set he does mention the spam rate, number of instances and number of features in the data set which all correlate to SpamBase. Idris work shows an ANN trained with the negative selection algorithm achieving 94% accuracy and notably 0.299% false positive rate, by far the lowest presented in related work.

The final paper reviewed is "Ham and Spam E-Mails Classification Using Machine Learning Techniques"(3). This paper explores the models seen in Figure 2.12. This paper shows random forest performing the best followed by artificial neural network and logistic regression. Most notably the random forest shown here is the best performing presented in terms of accuracy and precision. Upon further inspection of the figures in 2.12, the authors give the number of true positives, false positives, false negatives, and true negatives predictions who's values raise a question. These sum up to 4601 for each of the models, which is the number of instances of emails in the data set. This means their computed metrics for the models include prediction of feature vectors (emails) used when training the models. Due to overfitting a given model will always perform better on data it is trained with, meaning these reported results are inflated.

Table 3. Parameters of each of the classifiers used in identification of Spambase UCI dataset.

Classifier	Parameters	TP	FP	FN	TN	^{TP}rate	^{FP}rate
Random Forest	Trees = 100, Seed = 1	2713	75	134	1679	0.955	0.055
ANN	Iterations = 2000, lr = 0.3, mc = 0.2	2618	170	179	1634	0.924	0.084
Logistic	Max iterations = −1, ridge = 1*10−8	2645	143	206	1607	0.924	0.089
SVM	Kernel function = Polynomial	2651	137	236	1577	0.919	0.098
Random Tree	Min weight of instances in a leaf = 2	2592	196	191	1622	0.916	0.092
KNN	Linear Search with neighbors = 1	2585	203	221	1592	0.908	0.103
Decision Table	Based on Best First search	2663	125	321	1492	0.903	0.125
Bayes Net	Bayes search + Simple estimator	2620	168	301	1512	0.898	0.124
Naïve Bayes	Uses unsupervised Discretization	2621	167	300	1513	0.899	0.124
RBF	Cluster number = 2	2169	619	181	1632	0.826	0.148

Note. lr = Learning rate; mc : momentum constant.

Table 4. Seven measurements for spambase UCI classification.

Classifier	Precision	Recall	F-measure	Roc Area	Sensitivity	Specificity	Accuracy (%)
Random Forest	0.955	0.955	0.954	0.988	0.9529	0.9572	95.4575
ANN	0.924	0.924	0.924	0.958	0.9360	0.9085	92.4147
Logistic	0.924	0.924	0.924	0.971	0.9277	0.9182	92.4147
SVM	0.919	0.919	0.919	0.91	0.9182	0.9200	91.8931
Random Tree	0.916	0.916	0.916	0.922	0.9313	0.8921	91.5888
KNN	0.908	0.908	0.908	0.908	0.9212	0.8869	90.7846
Decision Table	0.904	0.903	0.902	0.948	0.8924	0.9226	90.3065
Bayes Net	0.898	0.898	0.898	0.965	0.8969	0.9	89.8066
Naïve Bayes	0.899	0.899	0.898	0.964	0.8973	0.9006	89.8500
RBF	0.845	0.826	0.828	0.9	0.9230	0.7250	82.6125

Figure 2.12: Results from (3)

3 Design

This chapter focuses on the design aspect of the research undertaken. Due to the focus being on the performance of different machine learning models each model was built individually. Once they were working the measurements were added in. The final application sequentially runs each of the models and outputs their performance metrics. The application was built for experimental purposes to investigate the hypothesis of this dissertation.

The differences the between models lies in the implementation of the algorithms looked at in the state of the art, but the overall structure of any system implementing these models would be the same. This allows us to look at one of the models in detail but gain an understanding how all the models were implemented.

The following sections cover each of the stages of building, learning, and evaluating a machine learning model.

3.1 Gathering Training/Testing Data

The first step in building a machine learning model is to gather your training/testing data. For supervised learning classification problems these must be labeled data sets. Whatever the underlying issue being evaluated is, the training set must be a matrix where each row represents the problem being modeled (i.e an email that may be spam) and the number of columns correspond to the features you are representing (57 in the case of the spambase data set), these columns holding numerical values only. There must be a corresponding class label vector that is 1 column wide which has numbers corresponding to each class of data (i.e 1 indicates email is spam, 0 indicates ham).

See the feature matrix and corresponding class labels below for an example of a training/testing set.

$$\begin{bmatrix} 1 & 5 & 0 & ..... & 43 & 1 \\ 6 & 23 & 53 & ..... & 3 & 41 \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ 134 & 523 & 23 & ..... & 4 & 2 \\ 5 & 1 & 3 & ..... & 599 & \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ . \\ . \\ . \\ 1 \\ 1 \end{bmatrix}$$

Typically when training a model, the training/testing data is gathered in one set and then randomly split into a training set and testing set. Normally around 70% of the data is used for training and the other 30% is used for testing (evaluating) the trained model. Or, as in this research when the size of the training set is small, cross validation can be used as outlined in Section 3.5.1.

3.2 Pre-Processing Training/Testing Data

After gathering the training and testing data pre-processing can then occur. While it is not absolutely necessary it can drastically improve the performance of a model.

Before training each of the models in this study the features were first normalised and then a bias dimension was added to the input. Normalisation scales the values of the features such that they are all within the same range but maintaining the differences in their original values. It is particularly effective when the data set has values that represent very different things and are in very different ranges. For example in spam detection, one feature may represent number of capital letters in an email and has a value of 121, another feature may represent percentage of occurrences of "free" in an email and may be 0.05. Not normalising these values would make the capital letter feature have a much bigger effect on what prediction a model makes.

The bias dimension (or node) is added to the input layer and always outputs a value of 1. This dimension gives flexibility to the model by effectively shifting the activation function to the right or left. Consider the case where all input features have a value of zero. The bias node (in combination with its positive or negative weight) ensures that the next layer produces a non-zero output. Bias provides the same functionality as a constant b in the linear function $y = mx + b$, ensuring that the line does not pass through the intercept (0,0) in the case where x is 0 and b is non zero.

3.3 Build model

The next section is the model building stage. In the case of the SVM or Softmax, this involves declaring the number of classes in the model and initializing the starting weights.

In the case of declaring the neural network there are a number of steps. Firstly the topology must be declared. This is the number of nodes in each of the layers in the system. After the topology is declared the activation and loss functions must be chosen. Finally, the weights for each node must be initialized to a random value.

3.4 Train model

After the data has been pre-processed and the model has been declared it is ready to be trained. The final parameters to be declared are the optimization method (e.g Gradient Descent, BFGS, etc) and the regularization term (L2/L1).

The training data is fed into the model in batches (size dependent on optimization method) which learns the weights such that the error in predictions is minimized across the training data. As the initial weights are random and there is the possibility of the optimisation methods to get stuck in a local minima, each of the models are trained a number of times and the best performing of these models is chosen. To ensure results are comparable, each model is allowed 100 iterations to adjust weights during 1 training run, and each model is trained 5 times with the optimal solution being chosen.

The trained model can then be used to make predictions on unseen feature vectors.

3.5 Evaluate Performance

The final stage in training a model is to evaluate its performance so that models can be compared, whether that be against other models of the same type but with different activation/loss functions or models of a completely different type. The labeled testing set is used for this.

To evaluate the performance of the models the measurements outlined in table 1.1 in the introduction are used. To calculate these (non time related measurements), the model makes a prediction for each of the vectors in the training set. These predictions can be used to calculate the number of true positive, true negative, false positive and false negative predictions in the set. From these a confusion matrix is built and the other measurements are calculated of this. The resulting measurements can then be used to compare these results.

For the time related measurements, each training/prediction being measured was run a number of times and then a statistical method called bootstrapping is used. This allows accuracy, variance and confidence intervals to be calculated for the timing measurements.

3.5.1 Cross Validation

To ensure the results were statistically significant 10 fold cross validation was used. Cross validation is a re-sampling method which tests how well a model generalises to unseen data after training, and it can identify when a model is prone to over-fitting the training data. It is particularly useful when the size of the training/testing set is not large.

10 fold cross validation works by splitting the training/testing set into 10 sub sets. Each model is trained 10 times. Each time 1 of the subsets is used for testing the model and the other 9 are used to train the model. These results can then be averaged giving a more accurate estimation of how a given model will respond to unseen data. In essence, cross validation allows the models to be evaluated on more unseen data (each of the 10 sets is unseen during training in 1 of the runs).

4 Implementation

This chapter outlines the implementation of the research including the main tools used and some examples of the source code. The application was developed and run on a Macbook pro with a 2.7 GHz Intel Duel Core i5 processor.

4.1 Haskell

Stack version 1.9.3 was used as the build management tool for the project. This eases the burden of managing packages which use different underling versions in their dependencies. Stack uses the Glasgow Haskell Compiler (GHC) to compile the source code. Dependency management as well as other configurations are managed through .yaml configuration files.

The resolver defines a set of packages (libraries) which are verified to work together to use as a base for the application. Packages needed outside of this are defined in the extra-dependencies field in the configuration file. The resolver is used as a base layer, with the extra project's dependencies being added on top of it. The resolver lts-9.21 was chosen for this project based off the required dependencies.

4.2 Machine Learning

When choosing a library to work with there was limited choice and some considerations to take into account. The first thing to consider was that the nature of the research includes covering a number of machine learning models. Secondly, the models should be written in purely Haskell. The "Data Haskell" group list the best library for machine learning applications. While there are a number of libraries that implement 1 model, the goal was to find one which had a number so that it could be

investigated properly as opposed to working with a number of libraries with shallow knowledge of how they work.

There were three potential choices. The first, which has the most support is bindings to tensor-flow. This is bindings from Haskell to tensor-flows underlying c++ implementation. This library had the most support when compared to the others. While some people claim that it is good for machine learning as you get the type safety and high level functional design from Haskell and efficient computations in c++, it did not fit the requirement of being a pure Haskell machine learning library.

The second option is called Hlearn and it looked like a good candidate. Its readme in its github repository claims it to be a "high performance machine learning library" with aims to be "as fast as low-level libraries written in C/C++" while also being "as flexible as libraries written in high level languages like Python/R/Matlab" (25). After looking further into it however, I found comments from the author on threads related to machine learning in Haskell saying that he has stopped development on it and recommends that no one uses it. He also claims that he stopped developing it due to Haskell's type system not being as flexible as he would like, and that he intends to return to the project in a few years time.

This left one final option called "mltool"(26). This library provides a number of supervised and unsupervised learning models as well as a small number of cost and activation functions. It could use some extensions in terms of the models it offers as well as more activation and loss functions. It is written entirely in Haskell and is currently the best library available in Haskell for machine learning. This library was chosen to implement the models after some experimentation with the other possible libraries.

4.3 Time measurements

To accurately measure the training and prediction times for the various models a Haskell benchmarking tool called Criterion(27) was used. It is the standard tool for benchmarking Haskell functions. It provides support for timing both IO and pure functions.

The code snippet in listing 1 shows an example of benchmarking a trained SVM making predictions on the matrix of training and testing features called xTraining and xTesting respectively.

	Predicted Ham	**Predicted Spam**
Actual Ham	True Negatives	False Positives
Actual Spam	False Negatives	True Positives

Figure 4.1: Confusion Matrix

Each of these function is executed between 10-10000 times and the resulting times are then performs bootstrapping to give a 95% confidence interval for the times. A nice feature of Criterion is that it can tell if the results are inflated due to outliers, and outputs the percentage inflation in results due to this. The underlying cause to this is usually other processes on the computer taking CPU time. Because of this measurement it was possible to rerun the results when they were inflated to ensure the final timing results were accurate.

```
1     defaultMain
2       [ bgroup
3           "Svm"
4           [ bench "training" $ nf (predictYSvm svm thetaSvm) xTraining
5           , bench "testing" $ nf (predictYSvm svm thetaSvm) xTesting
6           ]
```

Listing 1: Criterion Benchmark Example

4.4 Measurements

To evaluate the performance of the models, each of the predictions are classified into 4 categories. These are false positive, false negative, true positive and true negative classifications. These can be displayed in a confusion matrix in figure 4.1.

Listing 2 shows the code for calculating the number of false positive, false negative, true negative and true positive predictions respectively. These were then used to

calculate the performance measurements.

```
calculateFP :: T.Vector -> T.Vector -> T.R
calculateFP yExpected yPredicted = V.sum discrepancy
  where discrepancy = V.zipWith f yExpected yPredicted
        f y1 y2 = if round y2 - round y1 == 1 then 1 else 0

calculateFN :: T.Vector -> T.Vector -> T.R
calculateFN yExpected yPredicted = V.sum discrepancy
  where discrepancy = V.zipWith f yExpected yPredicted
        f y1 y2 = if round y2 - round y1 == -1 then 1 else 0

calculateTN :: T.Vector -> T.Vector -> T.R
calculateTN yExpected yPredicted = V.sum discrepancy
  where discrepancy = V.zipWith f yExpected yPredicted
        f y1 y2 = if (round y1 == 0) && (round y2 == 0)  then 1 else 0

calculateTP :: T.Vector -> T.Vector -> T.R
calculateTP yExpected yPredicted = V.sum discrepancy
  where discrepancy = V.zipWith f yExpected yPredicted
        f y1 y2 = if (round y1 == 1) && (round y2 == 1)  then 1 else 0
```

Listing 2: Classification Rates Calculations

4.4.1 Accuracy

Accuracy is the first measurement considered as it is the most intuitive. It is the percentage of correct predictions across the testing set. Accuracy is a good evaluation criteria when the number of positive and negative classes in the test set are equal (or close to). However, in the training set used here there are 814 Ham emails and 566 Spam emails. Other measurements are needed in this case to evaluate performance.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

4.4.2 False Positive Rate

False positive rate is the rate at which ham emails are classified as spam. This is a particularly important metric for spam filters as this is important to be kept at a minimum.

$$False\ Positive\ Rate = \frac{FP}{FP + TN}$$

4.4.3 Precision

Precision is the ratio of correctly predicted spam email to the total number of spam predictions. As the number of false positives should be as small as possible, a well performing model in that regard should have very high precision.

$$Precision = \frac{TP}{TP + FP}$$

4.4.4 Recall

Recall is the ratio of spam predictions to the actual total number of spam emails in the set. It is important that this is high but it is not as critical as precision as a few spam emails getting through the filter is acceptable.

$$Recall = \frac{TP}{TP + FN}$$

4.4.5 F1 Score

F1 score is the harmonic mean of recall and precision. Although it is not as intuitive as accuracy it provides a better overall indication of the performance as it not only takes into account false positives and false negatives and the ratios of them. F1 score gives an indication of which model is keeping both false positive and false negative rates low.

$$F1\ Score = \frac{2 \times (Precision \times Recall)}{Precision + Recall}$$

4.5 Model Parameters

The table below outlines the optimisation techniques, loss functions and other parameters used for each of the models trained. Note that L2 regularisation and normalisation of features is common across all models.

Model	Optimisation Technique	Loss function	Other
SVM	BFGS	Hinge Loss	N/A
SVM	GD	Hinge Loss	N/A
SM	BFGS	Cross Entropy Loss	N/A
SM	GD	Cross Entropy Loss	N/A
Logistic	BFGS	Least Squared Loss	N/A
Logistic	BFGS	Least Squared Loss	PCA
ANN	BFGS	Logistic Loss	Relu Activation Function. 58 nodes input later, two hidden layers with 50 nodes, 1 output layer with two nodes.

5 Results

5.1 Cross Validation

Each model was trained and evaluated 10 times to perform a 10 fold cross validation as described in 3.5.1. The results of each of the runs is used to calculate the performance measurements. These are then averaged so they can be compared across models. Section 5.3 contains the comparison of the averaged metrics. Figure 5.1 shows the results for 10 runs of the neural network as an example.

5.2 Time Measurements

This section covers the two time measurements taken. The first being model training time and the second being prediction times. These times correspond to 1 run of the cross validation, training on 4140 (90%) of emails in training set. Prediction times correspond to the cost in a stream computing application. Prediction time is measured for each model making predictions across the whole testing set, these results were scaled up to 1 million predictions for comparison purposes. As training times are done in advance of deployment, training times are include for comparative purposes only.

The Table 5.1 shows the mean training time as well as the lower bound (LB) and upper bound (UB) for the 95% confidence interval as calculated by the bootstrap method.

Neural Network	Accuracy	Precision	Recall	F1
R1	94.13043478	0.9371428571	0.9111111111	0.03928571429
R2	91.73913043	0.8770949721	0.9075144509	0.07665505226
R3	94.34782609	0.9289617486	0.9289617486	0.04693140794
R4	95.65217391	0.9255319149	0.9666666667	0.05
R5	93.91304348	0.9502762431	0.9005235602	0.03345724907
R6	94.7826087	0.9488636364	0.9175824176	0.03237410072
R7	94.56521739	0.917721519	0.923566879	0.04290429043
R8	95.43478261	0.943877551	0.9487179487	0.04150943396
R9	99.7826087	1	0.9948453608	0
R10	94.14316703	0.9171270718	0.9325842697	0.05300353357
Average	**94.84909931**	**0.9346597514**	**0.9332074413**	**0.04161207822**

Figure 5.1: Neural Network 10 Fold Cross Validation Results

	NN	SVM BFGS	SVM GD	SM BFGS	SM GD	Logistic Regression	Logistic Regression with PCA
LB 1 Mill Predictions (s)	5.6005	2.3042	2.5064	2.4451	2.5064	2.7485	2.4925
Ave 1 Mill Predictions (s)	5.6826	2.3592	2.5450	2.4975	2.5450	2.8052	2.5611
UB 1 Mill Predictions (s)	5.7707	2.4196	2.5940	2.5366	2.5940	2.8661	2.6263
LB Training Times (s)	17.46	14.9	11.83	7.556	14.98	2.647	5.498
Ave Training Times (s)	18.5	15.53	11.91	7.577	15.09	2.658	5.508
UB Training Times (s)	19.18	16.94	11.95	7.586	15.24	2.677	5.523

Table 5.1: Time Measurements

5.2.1 Training Times

The training Times in figure 5.2 show that the neural network takes the most time to train as expected at 18.5 seconds. This is followed by SVM with BFGS optimisation at 15.53 seconds and SM with gradient descent at 15.08 seconds. Interestingly the SVM trains quicker with gradient descent optimization where the softmax classifier trains quicker with BFGS optimization. The training time for logistic regression with PCA was about half of training without PCA, but the PCA itself took about 4 seconds. Logistic regression models train significantly quicker than the other models.

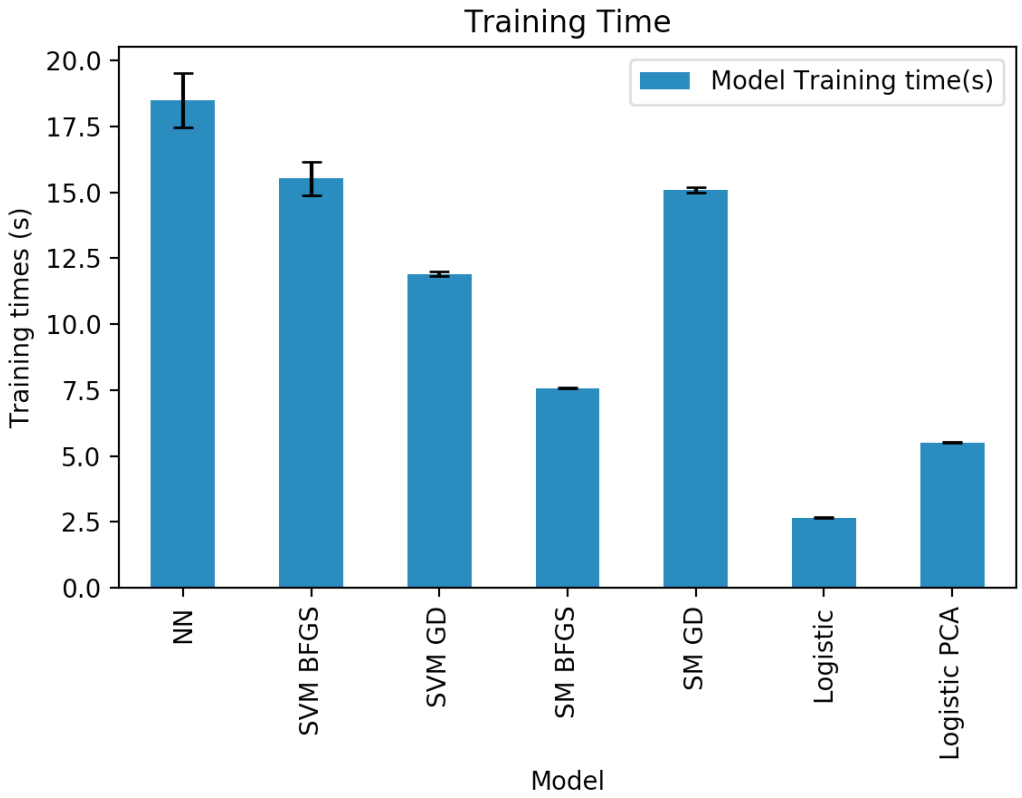

Figure 5.2: Training Times

5.2.2 Prediction Times

The prediction times in figure 5.3 show another expected trend with the neural network being at least twice as expensive when making a prediction. This is expected due to the increased complexity of this model.

The other times however are much closer than expected. The range in prediction times across the other modes is only 0.446 seconds (difference between logistic regression

and SVM with BFGS), however as this is across 1 million predictions, this would result in the SVM making 1,189,047 predictions against logistic regression making 1,000,000 which is somewhat significant in 2.8 seconds. Even slight variations in their performance would have a big impact on the number of correctly classified emails over aa given period of time.

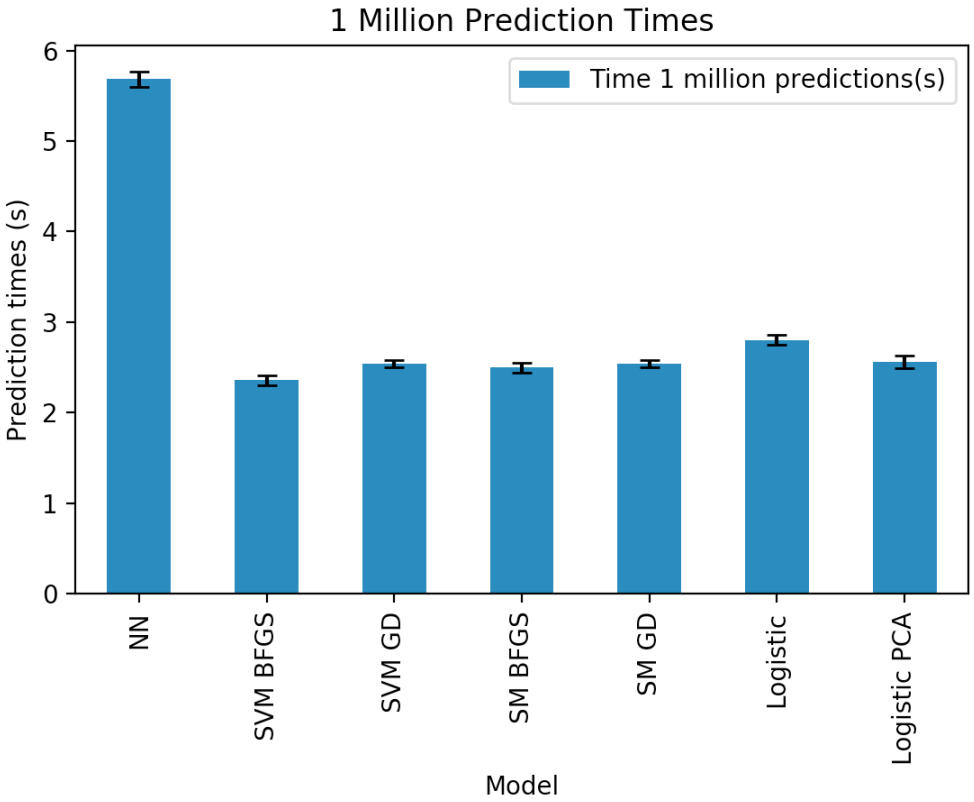

Figure 5.3: Prediction Times

5.3 Performance Measurements

Table 5.2 shows the performance measurements from the cross validation. Accuracy is calculated as a percentage. The other measurements are calculated as a ratio in the range of 0-1 as described section 4.4. The following sections identifies the key results of each measurement.

5.3.1 Accuracy

Figure 5.4 shows each of the accuracies plotted against prediction times. In terms of accuracy all models perform well. The neural network is marginally the best, closely

	NN	SVM BFGS	SVM GD	SM BFGS	SM GD	Logistic Regression	Logistic Regression with PCA
Accuracy (%)	94.84	94.69	92.61	94.65	90.9365	93.79	93.87
Precision	0.9346	0.9357	0.9431	0.9345	0.9355	0.9462	0.9459
Recall	0.9332	0.9282	0.8639	0.9282	0.8277	0.8931	0.8963
F1 Score	0.9339	0.9319	0.9018	0.9313	0.8783	0.9189	0.9204
False Positive Rate	0.0416	0.0410	0.0337	0.0417	0.0369	0.0330	0.0334

Table 5.2: Performance Measurements

followed by SVM and SM with BFGS. The range in results between the neural network, SVM and SM models with BFGS is less than 0.2%.

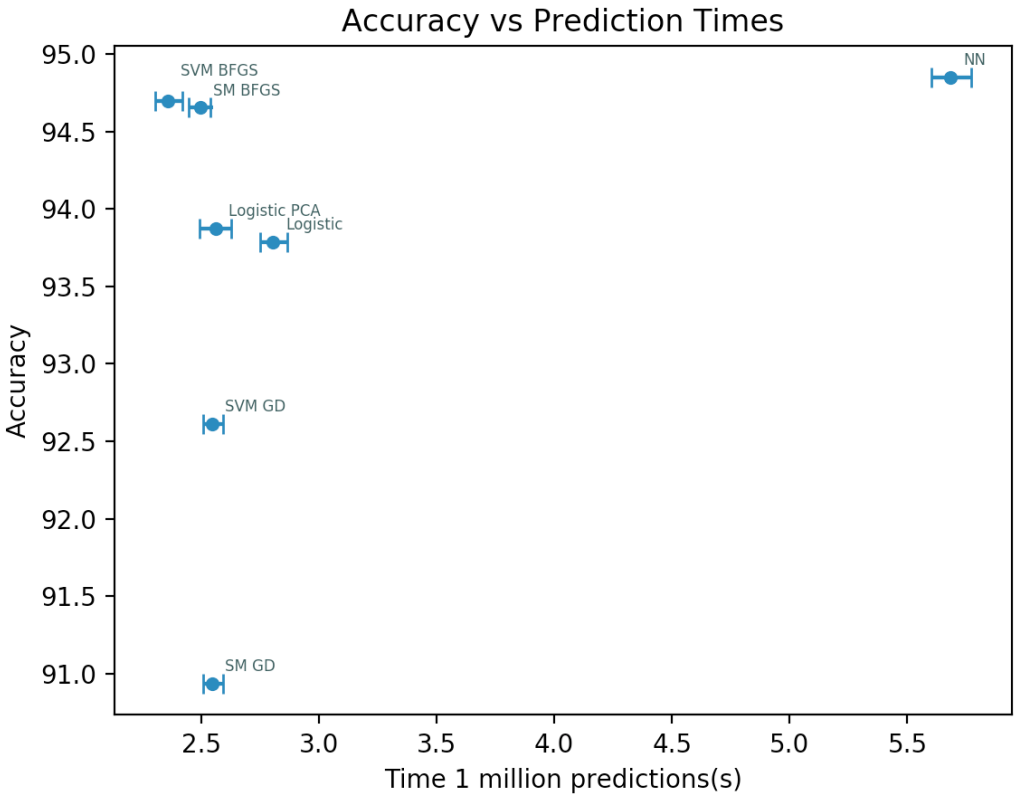

Figure 5.4: Accuracy

5.3.2 Precision

The precision results in figure 5.5 show an interesting results. Again all the models perform well. Logistic regression performs the best closely followed by logistic with

PCA and then SVM with GD. The neural network and SM with BFGS perform the worst. This shows the more straight forward model (logistic regression) performing the best as the most complex model (neural network) performing the worst in this context.

As precision corresponds to the ratio of true spam emails to predicted spam emails it is incredibly important this measurement is as high as possible. As any decrease in precision corresponds to more ham emails being classified as spam.

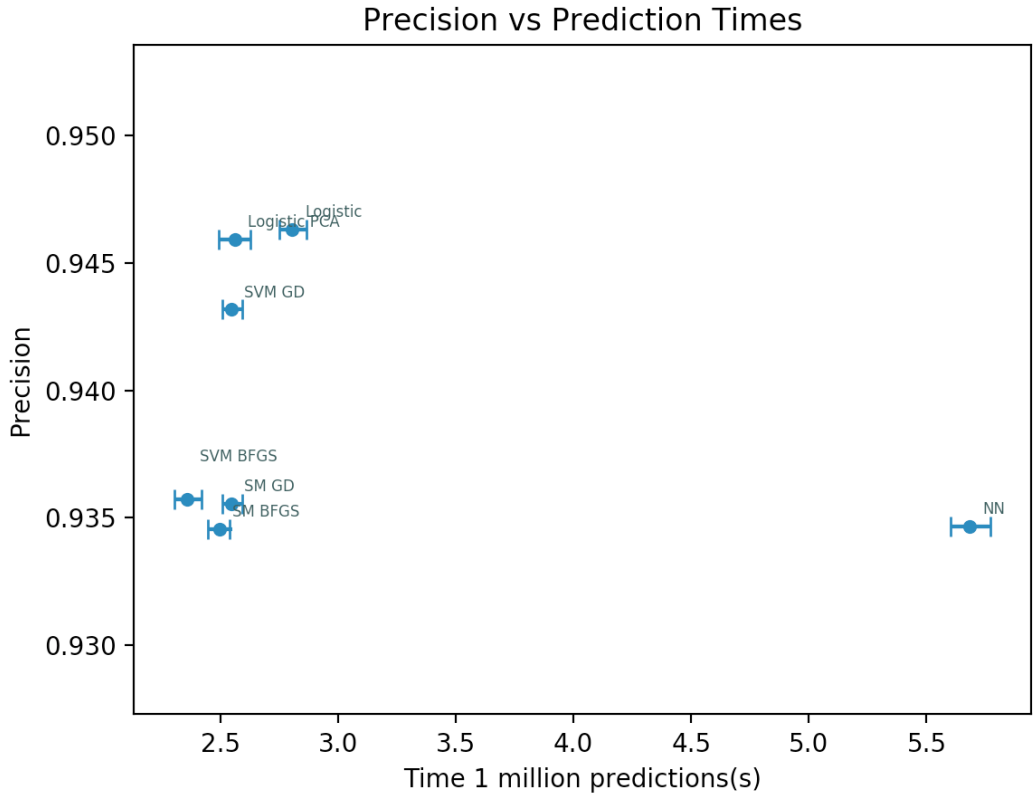

Figure 5.5: Precision

5.3.3 Recall

The recall results can be seen in figure 5.6. This shows a similar result as accuracy, with neural network performing the best followed by SVM and the SM with BFGS. In an ideal world recall should be as high as possible only in the case where there is no sacrifice to precision. Sacrificing recall for increased precision on the other hand is definitely an acceptable trade off in the context of spam detection.

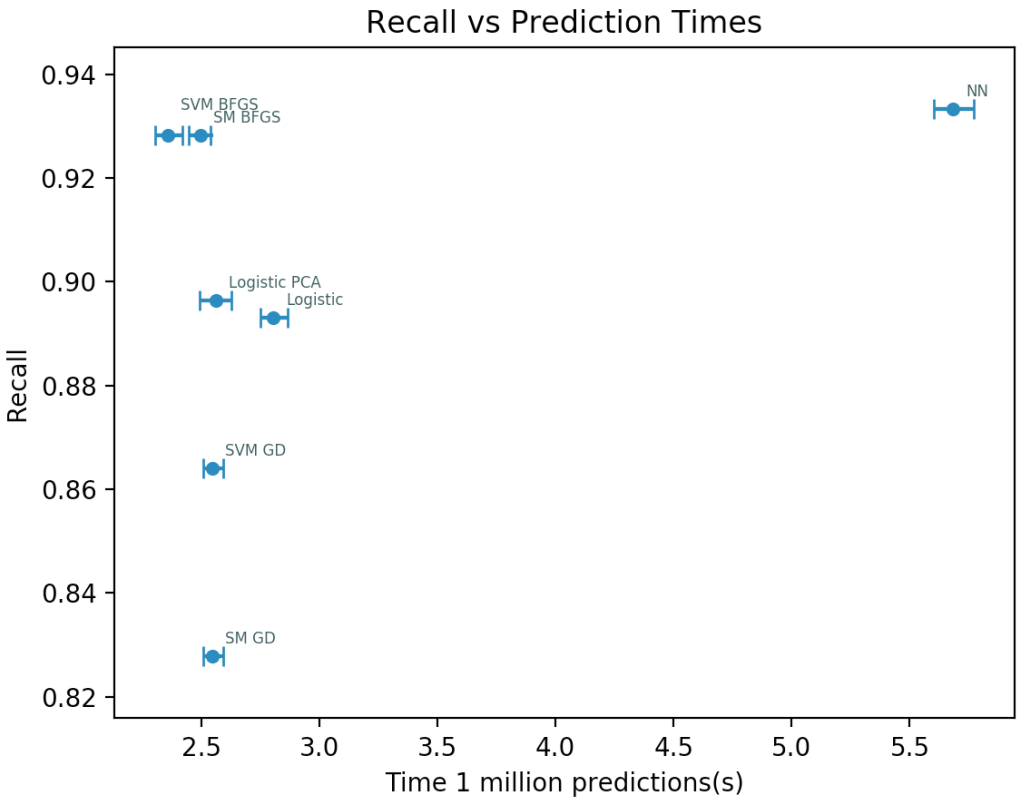

Figure 5.6: Recall

5.3.4 F1 Score

F1 scores in figure 5.7 show an incredibly similar graph to accuracy in figure 5.4. The models are ranked in the same order as in accuracy with Neural Network performing the best closely followed by tightly grouped SVM and SM with BFGS models.

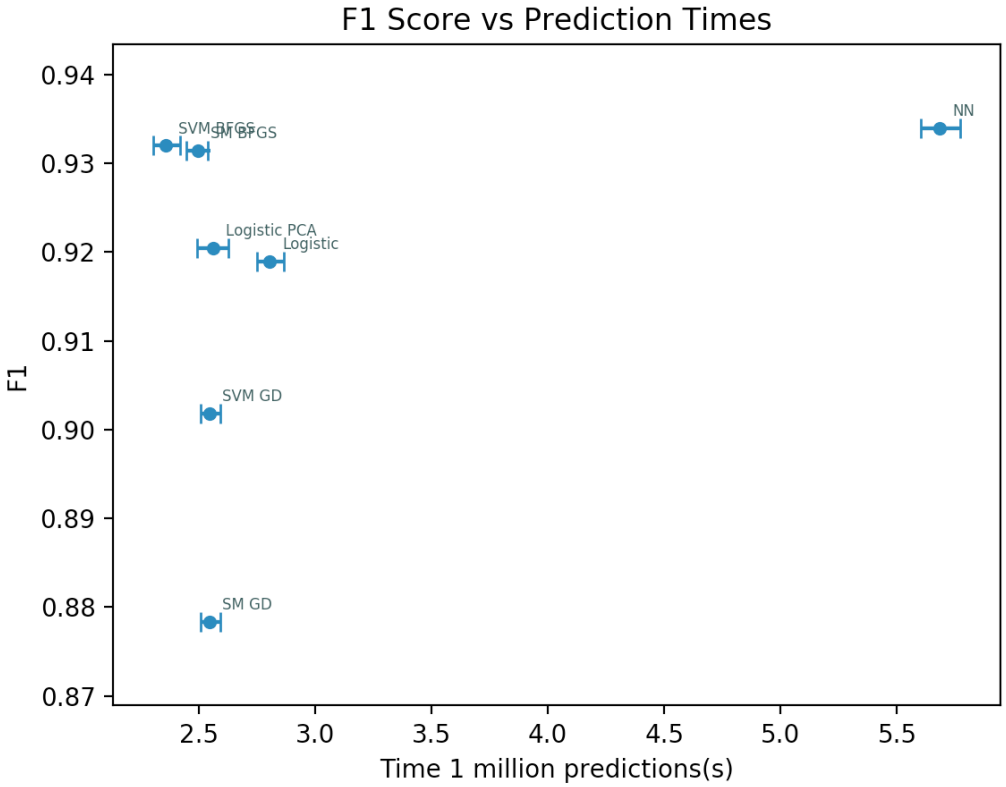

Figure 5.7: F1 Score

5.3.5 False Positive Rate

The final measurement can be seen in figure 5.8. As this directly correlates to the ratio of falsely predicted spam emails to the total number of actual ham emails, it is very important in spam detection and should be as low as possible. Logistic performs the best followed by logistic with PC and SVM with GD. The neural network performs the worst in this regard. Although the neural network correctly identifies the most emails overall, it also incorrectly classifies the most ham emails.

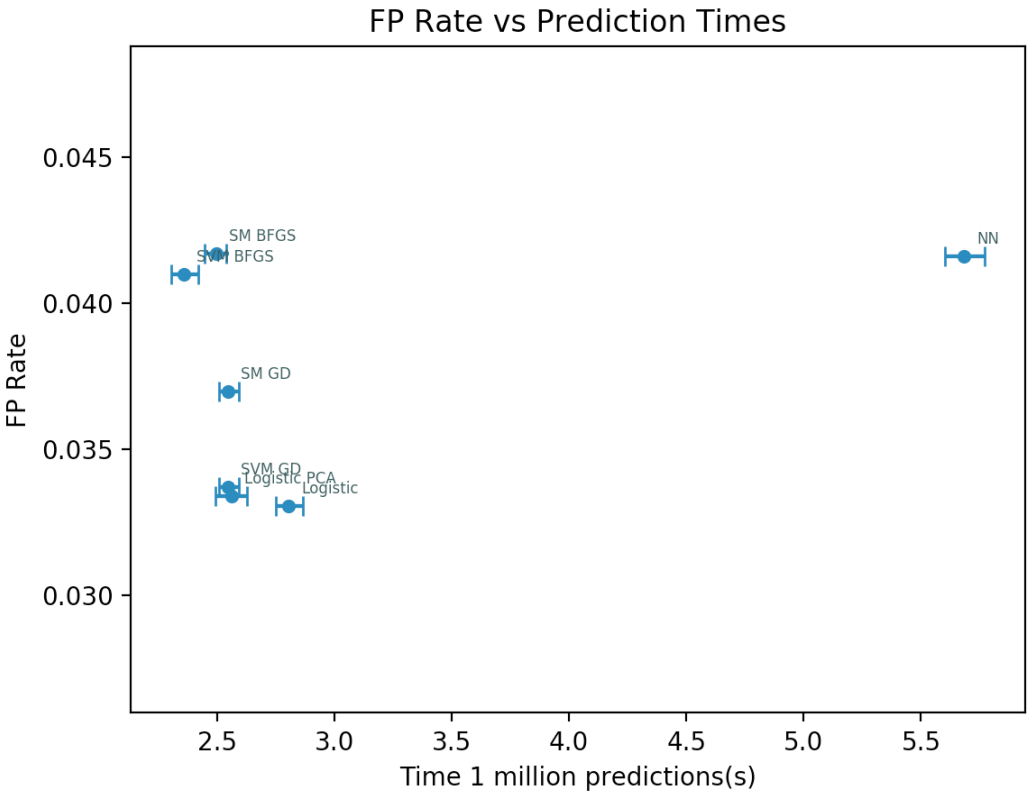

Figure 5.8: False Positive Rate

6 Evaluation

The results section show an interesting result. In regards to performance, in many binary classification contexts the neural network would be chosen as the best model as it tops both the accuracy and F1 score graphs by making the highest percentage of correct classifications. In the context of spam detection however, due to the fact the neural network is performing the worst in both precision and false positive rate it can be immediately ruled out when choosing a model to use. With the neural network ruled out the best performing model for spam detection would be the logistic regression models followed by the SVM with GD.

6.1 Is Algorithmic Adaption a Feasible Solution?

For Algorithmic Adaption to be a valid solution to load shedding in stream computing there are two trends that must be seen in the models evaluation. Firstly, there needs to be some variation in the prediction times of different model. Secondly, there must be a correlation between increased prediction time and models performance (i.e decreased false positive rate / increased precision).

In regards to the first trend, there is variation between different models prediction times. Most notably neural network taking over twice as long to make 1 million predictions when compared to the next quickest model. Leaving the neural network aside, the other models are within 0.446 seconds of each other. While this is quite close, this is for 1 million predictions and as such even slight variations in false positive predictions would have a massive impact on the number of incorrectly classified emails over a number of seconds.

Due to the neural network under performing the other models in the key performance criteria, while being the model with drastically increased prediction times, the proposed hypothesis of algorithmic adaption looks disproved under these conditions. To further

evaluate this though the best performing models need to be looked at in further detail, namely logistic regression and SVM with GD. Table 6.1 compares the predictions made 2.545 seconds when the system, has to deal with 1,102,239 emails, taking the rate of spam as the same as the rate of spam in spambase data set at 39.4%. This means there would be 669,958 actual ham emails and 434,282 actual spam emails.

Model	Spam emails in inbox	Ham emails in spam folder
Logistic Regression (load shedding excess)	46,424 (FP) + 40,282 (excess) = 86,706	22,108
SVM GD	58,280	22,577

If the system is overloaded such that it needs to process 1,102,239 emails in a given 2.545 seconds and logistic regression is used then 102,239 emails would be unprocessed (load shedded) and let into recipients inboxes. This would result in 28,426 extra spam emails being let into recipients inbox if the logistic model is continued to be used compared to the SVM GD model. In the same time period only an extra 469 ham emails would be incorrectly misclassified as spam. This matches the second trend needed for algorithmic adaption to be feasible.

With these results an argument could be made that the proposed hypothesis is in fact true as there is a trade of between filtering approximately an extra 32% of spam emails from inboxes incorrectly classifying approximately 2% extra ham emails as spam. However, due to the issue of false positives being critical in spam email and the prediction times of these models being quite close relatively speaking there is not enough evident to state the hypothesis is proved.

6.2 Comparison to previous work.

6.2.1 Performance

The previous work covered in 2.4.2 shows Saab's, Mitri's and Awad's(1) study finding an SVM to have accuracy and precision of 93.5% and 93.4% respectively beating their ANN model which achieved accuracy and precision of 94% and 92.9% respectively. These results show a comparable trend seen in this research with the neural network achieving the best accuracy but SVM beating neural network in terms of precision.

Sharma's and Arora's (2) paper showed a different story as it they presented a basic ANN achieving 93.28% accuracy but 94.332% precision, giving reason to believe the ANN could outperform other models in precision. This is not consistent with the results of this research. Arora's and Sharams's research, as with Saab's, Mitri's and Awad's research, present precision values of various models which all underperforming what one of the oldest classification techniques, logistic regression, was shown to perform in this research.

The comprehensive comparison of models in the paper "Ham and Spam E-Mails Classification Using Machine Learning Techniques"(3) have an interesting result. This research shows the random forest to perform the best. While there are some questions around their work as outlined in 2.4.2, the random forest could still be a good candidate for this data set.

In comparison to Idris' work (24), the results cannot be directly compared due to the different optimisation techniques explored. However Idris' work does show that there is the potential to increase the performance of the neural networks to above that of the other models. Due to it underperforming the logistic regression in this work it could not be considered for algorithmic adaption however if results similar to Idris' are possible it could have proved the proposed hypothesis to be correct.

In relation to the models evaluated, their performance in their Haskell implementations in this research was shown to be on par if not exceeding that seen in the related work. Excluding Idris' work, the other related works do not give enough information on the optimization methods, regularisation terms used, and if normalisation was performed on the data set. As such there is not enough information to evaluate if the differences seen are due to implementation details specific to the languages used or if its due to the chosen optimization techniques and pre-processing during development.

6.2.2 Cost

Prediction time is the cost parameter in terms of stream computing. In both the work referenced in this dissertation and all others reviewed during the research the only one that includes prediction related time measurements is Saab's, Mitri's and Awad's work. In general, there seems to be no work of note into either the theoretical or practical time complexities of machine learning models when looking at prediction times. There is some work in regards to training times but that is not of much relevance when looking at machine learning from a stream computing context.

In Saab's, Mitri's and Awad's work the only overlapping model which includes prediction times is the SVM. They title it "Testing time" in Figure 2.9. They do not specify if "Testing Time" is just the time for the model to make predictions on the testing set, or if it includes evaluating the model. Assuming that it is just the predictions, following that they performed a 10 fold cross validation, the assumption is made that these times correspond to making 4140 predictions (90% of data set). They cite the SVM testing time taking from between 0.1204 - 0.0581 seconds as seen in Figure 2.9. Based on the SVM with GD in this research making 1 million predictions in 2.5450 seconds, it would take approximately 0.0105 seconds for the model to make 4140 predictions. This is a significant increase in the time taken in Saab's, Mitri's and Awad's which was performed in MATLAB 2009b on a 2.4 GHz quad processor. Due to the lack of information on what exactly these time measurements include and the lack of reference to prediction times (theoretical or practical) in other research there is not enough evidence to evaluate how Haskell performs in this regard to other languages. If the assumptions made are true, the Haskell is outperforming Matlab implementations for these models. However as these are assumptions and are the only reference to prediction times in literature, so there is not enough evidence to make this claim yet.

7 Conclusion and Future Work

To be able to fully prove the hypothesis presented there are two possible approaches to future work of this research. The first would be to continue investigating the hypothesis with this data set and look at different optimisation techniques for neural networks, or include a random forest model as it has been shown to perform well on this data set. As presented in the related work on this data set, there is evidence to indicate that neural networks have the potential to beat all other models in terms of minimising false positives. If this result was achieved with the neural network, due to its increased false positive rate and big trade off in increased prediction times, it would likely prove the proposed hypothesis.

While that would be a valid result, the alternative approach is much more likely to produce a more significant result. The spambase data set was shown to be linearly separable due to models such as logistic regression and SVM (without kernel function) performing well at classifying the data. If a non linearly separable data set was used, then SVM with kernel function and more complex variations on neural networks such as convolutional neural networks or recurrent neural networks would have much higher performance and cost then the models looked at in this research. In particular, if recurrent neural networks performed well on a data set it is likely that networks of different depth are likely to perform and evaluate differently on top of SVM with kernel function performing adequately. Under these conditions it is likely that algorithmic adaption with a number of switches possible would be shown to be effective at degrading the performance of a stream computing system under load better than the alternative of load shedding.

In conclusion, this research has shown that the proposed hypothesis is plausible in this context but the real value of algorithmic adaption lies with more difficult applications (i.e non linear data sets). This research has also shown that Haskell is a good candidate for machine learning. Although machine learning in Haskell is currently more challenging than the more popular alternatives due to the lack of support, Haskell's high level abstractions and ability to catch many bugs at compile time combined with

Haskell's power when building highly scalable solutions make it a prime candidate for the types of highly distributed machine learning applications we are likely to see in the future.

Bibliography

[1] Mariette Awad, Salwa Saab, and Nicholas Mitri. Ham or spam? a comparative study for some content-based classification algorithms for email filtering. 04 2014. doi: 10.1109/MELCON.2014.6820574.

[2] S Sharma and A Arora. Adaptive approach for spam detection. *IJCSI International Journal of Computer Science Issues*, 10:23–26, 01 2013.

[3] M. Bassiouni, M. Ali, and E. A. El-Dahshan. Ham and spam e-mails classification using machine learning techniques. *Journal of Applied Security Research*, 13(3): 315, 2018. ISSN 19361610. URL `http://elib.tcd.ie/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edb&AN=129717057`.

[4] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`.

[5] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003. ISSN 1066-8888. doi: 10.1007/s00778-003-0095-z. URL `http://dx.doi.org/10.1007/s00778-003-0095-z`.

[6] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, 2005.

[7] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment, 2003. ISBN 0-12-722442-4. URL `http://dl.acm.org/citation.cfm?id=1315451.1315479`.

[8] N Tatbul, Ugur Cetintemel, Stan Zdonik, M Cherniack, and M Stonebraker. Load shedding on data streams. 01 2003.

[9] Nesime Tatbul. Qos-driven load shedding on data streams. volume 2490, pages 779–783, 11 2002. doi: 10.1007/3-540-36128-6_36.

[10] Simon Loesing, Martin Hentschel, Tim Kraska, and Donald Kossmann. Stormy: An elastic and highly available streaming service in the cloud. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, pages 55–60, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1143-4. doi: 10.1145/2320765.2320789. URL http://doi.acm.org/10.1145/2320765.2320789.

[11] David Guerin. *Computational Shedding in Stream Computing*. PhD thesis, University of Dublin, Trinity College, 2018.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[13] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. URL http://people.csail.mit.edu/lpk/papers/rl-survey.ps.

[14] W.A. Gardner. Learning characteristics of stochastic-gradient-descent algorithms: A general study, analysis, and critique. *Signal Processing*, 6(2):113 – 133, 1984. ISSN 0165-1684. doi: https://doi.org/10.1016/0165-1684(84)90013-6. URL http://www.sciencedirect.com/science/article/pii/0165168484900136.

[15] J.L. Morales. A numerical study of limited memory bfgs methods. *Applied Mathematics Letters*, 15(4):481 – 487, 2002. ISSN 0893-9659. doi: https://doi.org/10.1016/S0893-9659(01)00162-8. URL http://www.sciencedirect.com/science/article/pii/S0893965901001628.

[16] Yichuan Tang. Deep learning using linear support vector machines. In *In ICML*, 2013.

[17] J.S. Cramer. The Origins of Logistic Regression. Tinbergen Institute Discussion Papers 02-119/4, Tinbergen Institute, December 2002. URL https://ideas.repec.org/p/tin/wpaper/20020119.html.

[18] J. A. K. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural Process. Lett.*, 9(3):293–300, June 1999. ISSN 1370-4621. doi: 10.1023/A:1018628609742. URL https://doi.org/10.1023/A:1018628609742.

[19] Warren S. McCulloch and Walter Pitts. Neurocomputing: Foundations of research. chapter A Logical Calculus of the Ideas Immanent in Nervous Activity, pages 15–27. MIT Press, Cambridge, MA, USA, 1988. ISBN 0-262-01097-6. URL http://dl.acm.org/citation.cfm?id=65669.104377.

[20] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[21] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[23] Data haskell - organization devoted to enabling reliable and reproducible data science and machine learning by leveraging the haskell programming language. https://www.datahaskell.org/. Accessed: 2019-04-09.

[24] Ismaila Idris. Spam classification with artificial neural network and negative selection algorithm. 2011.

[25] Izbicki Mike. Hlearn. https://github.com/mikeizbicki/HLearn, 2016.

[26] Ignatyev Alexander. mltool. https://github.com/aligusnet/mltool, 2018.

[27] Criterion - haskell benchmarking tool. https://hackage.haskell.org/package/criterion. Accessed: 2019-04-09.