Imperial College London
Department of Computing

BEng Computing Final Year Individual Project Report

Can you Poison a Machine Learning Algorithm?

Author:
Vasin Wongrassamee

Supervisor:
Dr. Luis Muñoz-González

June 19, 2017

Acknowledgements

I would like to thank my supervisor, Dr. Luis Muñoz-González, for his expert advice and wholehearted support throughout this project. His passion for machine learning has inspired me more times than I can count.

Contents

Chapter 1

Introduction

1.1 The Problem

With the advancement in computing capabilities, statistical machine learning has become a practical and effective solution for many large-scale decision problems in recent years. Coupled with the rise of internet-based services, online statistical machine learning has never been more widely used than today. To demonstrate how much impact this new technique has made to a person's life, here is a list of some online services that uses the machine learning algorithm: email spam filters [29], Google search engine ranking by relevance, Google drive's anti-viruses and malware filters, recommendation systems such as Amazon or Netflix suggestions [32]. There are many more applications of the machine learning algorithms in various fields such as computational biology [3], computational finance [38], and many others [44]. Having the machine learning system at the core of many applications gives malicious users (attackers) opportunities to compromise the systems by attacking their machine learning component, in turn, gaining various advantages and possibly profits. A new research field called *Adversarial Machine Learning* has emerged to study the vulnerabilities of machine learning systems in adversarial settings. However, since using machine learning in products or services is relatively new, its vulnerabilities in the real-world settings have not been fully explored. Attacks on the learning algorithm have already been reported in the wild [14],[12],[16], emphasising the relevance of this problem.

As long as the safety of the machine learning system remains questioned, the applications of machine learning in real-life systems will be limited. The fact that most machine learning systems require re-training on new real-world data periodically has allowed them to be adaptive when solving different problems, making them more superior than ordinary programs. However, it also means that users are essentially the ones who create the input to the dataset used to re-train the system, making the poisoning attack – an attack where malicious samples are injected into the learning system's training data – one of the most relevant attacks in practice. This work will explore this particular type of attack and shed light on the learning systems' vulnerabilities, allowing further studies to carry out in the hope of arriving at safe machine learning systems.

1.2 Contributions

In this work, I have contributed to the literature as follow:

1. I have explained in detail (with proof) the method an attacker can use to craft an *optimal* malicious training sample to inject into the training dataset in order to carry out the poisoning attack.

2. I have shown that the standard way [29] [28], as used in the literature, of crafting the poisoning sample is inefficient and unstable, and have proposed a new method of computing the poisoning points that is more stable and more efficient, inspired by the work of Andrew Ng et al. [9]

3. I have explored the novel back-gradient method to compute the poisoning points without having to assume a KKT condition assumption like in the other methods. This allows the poisoning attack to be carried out on a neural-network learning system. With this, I have investigated the effect of the poisoning attack on a Multi-Layered Perceptrons (neural network) classifier.

4. As the back-gradient method is now believed to be the most efficient way to solve the problem, I have examined and carried out experiments on the time-complexity of this method. I have also provided experimental comparisons of this method with the method I have proposed earlier, experimenting with 3 different real datasets – simulating 3 real-world applications.

5. I have introduced 2 attack strategies – greedy and non-greedy [2] –, and have provided the results of experiments to compare the effectiveness the two attacks on both linear and non-linear machine learning classifier systems, using 3 different real datasets – simulating 3 real-world applications.

6. I have extended the literature by applying the optimal poisoning attacks on a multi-class classifier – a work that has not been done before. I have explored both targeted and indiscriminate attack on the multi-class classifier and show the differences in the effect each of them has on the classifier. I have provided the results of this experiment in this work, and have used the 3 different real dataset which simulate the different real-world applications.

1.3 Report Structure

This report will explain, in detail, how attackers could use different methods to carry out the optimal poisoning attack on different well-known machine learning classifiers. It will also evaluate whether the attacks are effective, or in other word, harmful. The background chapter will describe the knowledge that is helpful to the understanding of this report, and the project-setup chapter will describe how the project and experiments are set up. After that, the report will follow the format of 'method discussion followed by relevant experiments and results'. The parameters used in each experiment will be included in the Appendix section so that the results can be re-created in the future if needed.

Chapter 2

Background

2.1 Machine Learning

A machine learning system is system that is able to *learn* from its experience. Learning in this context means finding the 'right' parameters. Therefore with this automatically adjustable parameters, the learning system is adaptive to changes in the real world - provided that it is re-train regularly with real-world data. Such systems are useful for two main tasks: solving regression and classification problems. In the *training phase* a machine learns from the *labeled* training samples i.e. sample input with a given answer. A trained machine would be able to derive a reasonable output answer from any future inputs. This stage is called the *inference* phase.

2.1.1 Machine Learning Tasks

The problems that the machine learning systems are designed to solve are divided into two classes: *regression* and *classification* problems. Both problems take in a set of input features values. An estimation problem produces a continuous output, and a decision produces a discrete output from a set of finite alternatives. An example of the regression problem would be predicting a housing price given the land area of the property. An example of the classification problem would be predicting whether a tumor is malignant or benign given the size of the tumor. The features in these cases are the area of land and the size of tumor [30].

2.1.2 Supervised Learning Model Representation

There are three types of machine learning systems: *supervised* learning, *unsupervised* learning, and *reinforcement* learning [26]. In this project, I have focused on the supervised learning, as it is perhaps the most popular approach in machine learning design. This section will describe the basic intuition behind the machine learning algorithm. A learning algorithm of a system takes in a set of training data and obtain a *hypothesis function* (h). This function is used in the *inference phase* to map the input features values to the answer of either a classification or regression problem.

In order to visualise it, the optimal hypothesis function for a regression task is the best fit line for the data points(both training and inference data), and the hypothesis function of a classification task would be the best line that separate between the different classes of data points.

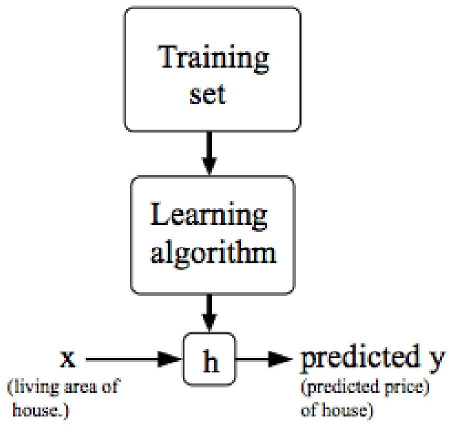


Figure 2.1: This figure illustrates the model of supervised learning, where the vertical direction is the *training phase*, and the horizontal direction is the *inference phase*

2.1.3 Deriving the Hypothesis Function

In cases such as in figure 2.2, it could be easy for human to estimate the hypothesis function i.e. the best fit line. However, this is only true for data points with 1 or 2 feature dimension. In figure 2.2 the feature spaces of the data points only have two dimensions. This means that the input data only has two features, or more specifically, only two features from the input data are used in the learning algorithm. In reality, machine learning algorithms use many more features than that. This makes it impossible for human to derive the hypothesis function by eye.

Most learning systems learn the 'optimal' hypothesis function in a different way from what a human would do. Instead of looking at all training data points at once to estimate the 'best fit' or 'best class separation' line, they take in one training input sample at a time and adjust the parameters of their hypothesis functions accordingly using the *cost function.*

2.1.4 Cost Function

Cost function or *Loss function* is the measurement of how far away from the correct *answer* the machine's answer - output of the hypothesis function - is. A typical example of a cost function would be the squared loss:

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=1}^{m}(h_{\Theta}(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{2.1}$$

where vector $\boldsymbol{\theta}$ is the current parameters of the hypothesis function; h_θ is hypothesis function with its parameters equal to $\boldsymbol{\theta}$; m is the number of training samples; $\mathbf{x}^{(i)}$ is the feature values of the ith training point; and $y^{(i)}$ is the *label* i.e. the correct answer of the ith training data point.

It is obvious that the higher cost means worse performance, and lower cost means better performance. In the training phase, cost function is calculated for a set of parameter values $\boldsymbol{\theta}$, then the *gradient descent* will calculate the adjustments in $\boldsymbol{\theta}$ which would minimise the cost function.

In the extreme case, when the cost is zero, it means that the machine has predicted the answer for all of the training data points correctly. One might think that the machine's hypothesis function is most correct in this case. However, it is almost impossible for the such hypothesis function to be the optimal one, as for a function to give zero cost it would mean that all the training data points lie perfectly on the hypothesis function (for a regression problem) or separated perfectly by the hypothesis function (for a classification problem). With the random nature of the real-world data, it would be naive to conclude that the optimal hypothesis function is found when its calculated cost is zero. The zero cost hypothesis function instead suggests that *over-fitting* happens. This problem of over-fitting will be explained later in section 2.1.6.

2.1.5 Gradient Descent

Gradient descent is a well-known algorithm that many learning machines use to find the optimal parameter values. As suggested by its name, the algorithm looks at the gradient of the cost function against each feature x_j of the input; i.e. the rate (or direction) of change of the cost value with respect to the change (increase) in the value of the feature, which is the derivative of the cost function with respect to an input feature $\frac{\partial J(\theta)}{\partial x_j}$.

```
repeat until convergence: {
    $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$        for j := 0...n
}
```

The algorithm above [30] shows how gradient descent is performed in the system discussed in the previous section where the cost function is the equation 2.1. Here, n is the total number of features and α is the learning rate - a user-controlled variable that is used to control the rate of learning.

The gradient $\frac{\partial J(\theta)}{\partial x_j}$ of the equation 2.1 is $\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$. To move closer to the minimal point, this gradient is used to update the parameter values θ_j corresponded to each feature j as seen in the algorithm above. Eventually, the algorithm will converge as near the minimal point the gradient will converge to zero. At this point, if the cost function is *convex* we can conclude that we have reached the optimal solution. It is worth noting if the learning rate α is chosen to be too large the algorithm might step pass the minimal point and not converge. For the *non-convex* cost function, the gradient descent might converge to a local minima. Increasing the value of the learning rate, as well as using more advanced techniques such as using *momentum* [37], may help avoid this problem.

2.1.6 Over-Fitting

The problem of over-fitting may arise when the system *tries too hard* to make sure that the cost of the hypothesis function is zero; i.e. generating a hypothesis function that is too complex and unrealistic in order to perfectly pass through or separate all the training data points. By having such complex function, the system loses the knowledge about the generality of its data, i.e. the best fit line would not be smooth. This knowledge about the data generality is essential for a realistic interpolation of the of the training data points, and therefore an over-fitted system would not be able to solve its tasks correctly.

Over-fitting could be avoid by discarding irrelevant features; letting the machine learn only the useful features. This process of choosing the right features i.e. *feature selection* is not easy, and it is an area that is being of research [21] [17]. In the case where all features are slightly useful, regularisation can be used to smoothen the hypothesis function by reducing magnitude of all parameters.

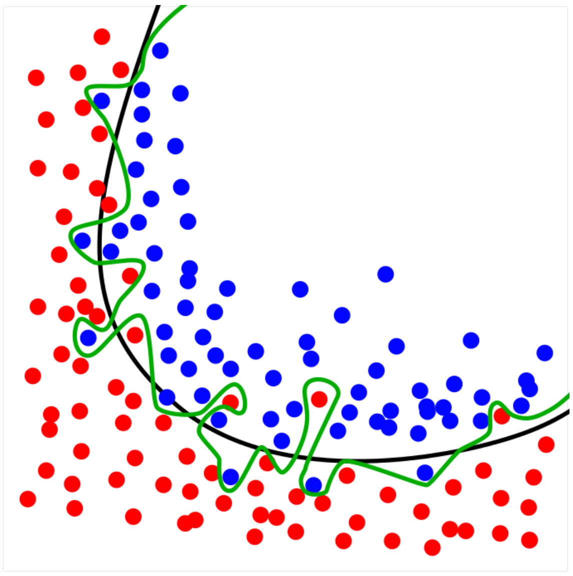


Figure 2.2: This figure illustrates an example of the problem of over-fitting in the classification problem, where the black line is the desire hypothesis function and the green line is the hypothesis function we might have when we experience over-fitting

2.2 Solving the Classification Problem

This project focuses on classification problems, because many researches in Adversarial Machine Learning involve the learning machine whose task is to classify an input; for example, email spam filter and malware detection. However, the knowledge gain from this project can be apply to the regression algorithm interchangeably. Here are some important machine learning classifier that I have come across in my research for this project.

2.2.1 Linear Classifiers

Perceptron and ADALINE

ADALINE [1] is a single layer neural network developed at Standford University in 1960. It is a single layer neural network where all of the features has a weight - similar to the parameter θ discussed before - and the output, 0 or 1, comes from the following formula:

$$y = sign(\boldsymbol{w}^T \cdot \boldsymbol{x} + b) \tag{2.2}$$

where vector w is the weights of each feature, vector x is the values of each feature, b represents the bias constant, and the *sign* function represents the quantizer.

The *perceptron* [42] works the same way as ADALINE, the difference is that ADALINE learns from the output value $\boldsymbol{w}^T \cdot \boldsymbol{x}$ – the pre-quantized value–, whereas the perceptron learns from the output value y – the post-quantized value. By learning from y the perceptron might encounter a convergence problem with its gradient descent algorithm since the discrete property of y causes the cost function to also be non-continuous.

Logistic Regression

Logistic regression is one of the most widely used learning algorithms today. One way to solve a classification problem is to turn it into a regression problem by adding an extra dimension z to the data point (i.e. the *y-axis* of figure 2.3). This z variable represents the classes of the data point: 1 being a positive class and 0 being a negative class. After we solve the regression problem, we could use the hypothesis function to predict the value z from the input data. By the nature of a regression problem, z would be a continuous value, and we would turn it into a discrete value by comparing it with a *threshold value* which we defined. If the value z is greater than the threshold it would be classified as positive, otherwise as negative. This discrete value, 0 and 1, is represented by y.

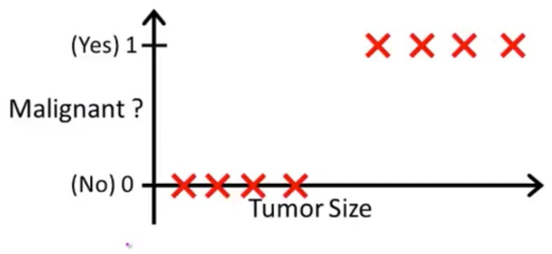


Figure 2.3: This figure illustrates a graph used to visualise the classification problem as a regression problem - a technique used in logistic regression classification.

As seen in figure 2.3, it is not suitable for the hypothesis to be a straight line. Therefore, the simple linear regression model would not work for a classification problem. The sigmoid function in logistic regression allows us to better capture the hypothesis function of a classification problem. Therefore logistic regression can be used as a classifier.

$$h_\theta(\boldsymbol{x}) = g(\boldsymbol{\theta}^T \boldsymbol{x}) \tag{2.3}$$

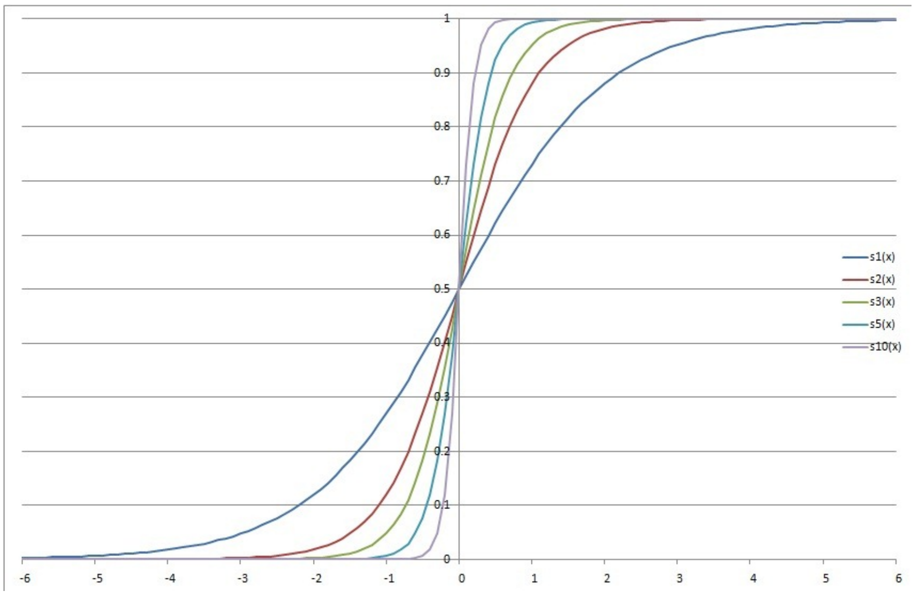


Figure 2.4: This figure illustrates examples of sigmoid functions

where $\boldsymbol{x}$ is the vector of feature values, $\boldsymbol{\theta}$ is the vector of parameters, and g is the sigmoid function defined as:

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2.4}$$

Logistic regression classifier uses gradient descent to learn the optimal parameter.

2.2.2 Neural Networks as Non-Linear Classifiers

Non-linear classifiers offer more sophisticated algorithms to classify data that are not linearly separable. An example of this is the famous XOR classification problem [20], which could be solved with a multi-layer neural network classifier [20] [30]. Here are some important and well-known multilayer neural networks.

Multi-Layered Perceptron

Many single-layered perceptrons can be combined to form layers of multiple-layered perceptron [15]. This allows it to perform a much more complex tasks of classification or regression. The model is trained using back propagation - a way to train the feed-forward neural network [11] by propagating the errors from the layers closest to the output back to the layer closest to the input, using those propagated errors to update the weights of each layer.

CNN

The Convolutional Neural Nets (CNN) are a feed-forward neural network that works well for tasks such as image or speech processing. The system uses filters which slide over all parts of the input and process the outputs for each section that it has passed through. The parameters that this system learn are the weight of each filter element. With this design, the weights are shared among the filtered sections of the input space, which is helpful when processing data that has similar features such as edges in images pixels [23].

Auto-Enconders

Auto-Enconders are used for unsupervised learning. The design is similar having a multi-layered perceptron connected to its reversed self at the output layer. This design allows the machine to perform unsupervised learning by setting the input as a target, and train the reversed part with back propagation. So the system is trained to approximate the identity function. When the training is done, the reversed half of the system can be removed, and the feed-forward half could be used to solve the classification problem. [41]

2.3 Adversarial Machine Learning Case Study

Before going into detail about Adversarial Machine Learning, this section will give a case study of the *SpamBayes* email spam filter [19] to demonstrate how a machine learning system is vulnerable to adversaries, and how an attacker might take advantages of the nature the machine learning algorithm in *SpamBayes*.

2.3.1 Classification

SpamBayes filter classifies emails using Bayesian Classifier. It extracts the email content and use the presence of each word (token) to determine the class - *spam*, or *ham* i.e. legitimate, or *unsure* - of the email [35]. Each word (token) has a *spam score* which is used to calculate the overall *message score*. An email with high *message score* will be classified as a spam, i.e. when an email contains many words that have high *spam score*, the email is likely to be classified as spam. Nelson et al. and Tsai explain in more detail the SpamBayes classification mechanism [29] [39].

In the training phase, when SpamBayes *sees* an email of *spam* class, it will increase the *spam score* for all of the tokens present in the email. Similarly, when the system sees a *ham* (legitimate) email, all of the token present will have its score reduced.

2.3.2 Attacks

SpamBayes retrains itself periodically with the new data that it has collected. This has created a way for an attacker to place his malicious data (email) into the training data set of the SpamBayes filter. Simply by sending the malicious email into the target user's inbox would cause the user to label it as a *spam* and, later, when the machine retrains itself the malicious email can become part of the training dataset that it would train from.

Knowing the classification mechanism and the training algorithm of SpamBayes, attackers can craft some malicious emails to attack the classifier; making the overall classification performance so poor that the whole classification system becomes useless with the *dictionary attack*, or making it unable to classify a certain email properly i.e. causing a wrong classification of a particular email with the *targeted attack*.

Dictionary Attack

The dictionary attack aims to render the SpamBayes filter useless by causing it to classify all emails - spam or not - as spam emails. The attack is done simply by including every word in a dictionary into an email. Such email will be recognised by the user as a spam, and if, later, the SpamBayes filter retrains itself with this malicious email, every word in the dictionary will have a higher *spam score*. Enough proportion of such dictionary email in the training set would break the SpamBayes filter.

Targeted Attack

The motivation of this attack is to prevent the target user from receiving a particular email i.e. *target email*. An assumption is made that the attacker have a good knowledge about the *target email*. This can be achieved by crafting a spam email that has all of the words that appear in the *target email*. When the system is trained with this data point, it would increase the score of all the tokens presented in the *target email*. In the future when the user receive the *target email* SpamBayes would classify it as a spam. This attack is also effective even if the attacker knows partially tge targeted email.

2.4 Studies of Adversaries in Machine Learning

2.4.1 Taxonomy

It is important for a research purpose to know how adversaries are classified and described by the researchers in the field. This taxonomy, proposed in [4], is used as the formal classification of adversaries in the field of computer security. Its usage is extended into the field of adversarial machine learning [19]. The taxonomy consists of three axes: *influence*, *security violation*, and *specificity*, which can be used to classify any form of attacks against a machine learning system.

Influence

This axis of the taxonomy measures the degree which an attack can influence the learning algorithm. An attack can be categorised into two types – *causative* and *exploratory* – depending on its capability to influence the targeted learning system.

- ***Causative***: This type of attack will modify the learnt model of the targeted system to cause it to behave differently i.e. will break the system or service, for example this kind of attack can cause an email spam filter software to classify a spam email as non-spam email or vice-versa.

- ***Exploratory***: This type of attack cannot alter the learning system's behaviour. Hence the aim of such attacks is, usually, to explore the targeted system to gain information and understanding on the system's vulnerabilities. With good understanding of the systems weaknesses, attackers can then carry out evasion attacks. For example, attackers could craft malwares that are able to avoid detection from the targeted malware detection system.

Security Violation

This axis of the taxonomy categorises the type of security violations that an attack can cause to the system.

- ***Integrity***: These attacks focuses on causing the system to produce incorrect results. When targeting a classifier system, this attack would try to increase false-negative classifications. For example, it would allow malwares to pass through the targeted malware filters.
- ***Availability***: These attacks aim to causes the targeted system to produce incorrect results in any possible ways i.e. both false-negatives and false-positives. The objective of this type of attack is different from the *integrity* case, as *availability* attacks aim to make the system unusable for the users.
- ***Privacy***: These types of attacks aim to obtain confidential information of the users of the targeted services. These attacks are not yet the main concerns to the machine learning systems.

Specificity

This axis of the taxonomy categorises the intention of the attackers when attacking the targeted system.

- ***Targeted***: These attacks aim to degrade a specific point of a system. For example, it would cause a spam filter to classify a particular legitimate email as a spam email. This type of attacks often require more knowledge on the training data and training algorithm, as the attack has to be very specially crafted in order to target a specific point or sample.
- ***Indiscriminate***: These attacks aim to degrade the overall performance of a system. This type of attacks usually require less data on the training data and training algorithm.

In theory, attacks from any categories in each axis in the taxonomy is possible, however, the two most relevant attacks in practice are *exploratory integrity attacks* i.e. the evasion attack, and *causative availability or integrity attacks* i.e. the poisoning attack. In this work, we will focus on the poisoning attack.

2.4.2 Attacker's Capabilities in Causative Attack

Having a formal model to measure the capability of a causative attack can be useful when it comes to evaluation or comparison of attacks.

Corruption Model

Corruption models [19] are used to evaluate the capability of adversaries by analysing their restrictions. Below describes the two corruption models and when to use each one.

- *Insertion Model*: This model measures the capability of an attack by the amount of attacking data points (number of the poisoning data) needed in order for the attack to be effective. This model is effective when measuring the capability of an attack from an attacker who is only allowed to modify a limited amount of training data points. The content or features of the data points, however, can be modified arbitrarily. An example of such attacks is the email spammer attack, where the attacker can craft the malicious emails in any ways he likes, but only able to send a limited number of those emails to the training algorithm of the spam filter.

- *Alteration Model*: This model measures the capability of an attack by how much a data point is altered. It assumes that there is a limited amount of alteration that can be done on each data, but the number of data points being modified can be arbitrary.

Class Limitation

We can look at the *class limitation* [19] of an attack to analyse its capability, i.e. looking at which class data – negative, or positive, or both – an attacker is allowed to create or poison. In the spam filter example, the attacker can manipulate both classes in order to create an effective attack. Positive (malicious) class is done to bring the score of some words down so that legitimate emails with those word will later be classified as a spam. Negative (legitimate) class is done to bring the score of some words up so that the spam email with those words can be classified as legitimate later, however, this case is unlikely to happen in practice. It is worth noting that not all systems allow the attackers to alter both positive and negative classes of the training data. In fact, it is usually the case that the attackers are limited to manipulate only the positive (malicious) class of the data. For example, in spam filtering applications, it is reasonable to assume that the malicious emails are labeled as spam.

2.4.3 Attacker's Knowledge

The attackers will have a certain degree of knowledge about the learning system. Three main types of knowledge that are critical to attacking a machine learning system are the knowledge of the learning algorithm, knowledge of the feature space, and the knowledge of data. One might be tempted to keep all of these information secrets, as that would result attacker having no knowledge of the system. However, like any security system, Kerckhoff's Principle suggests against over relying on secrecy; as secrets can, one way or another, be exposed and when that happens all other components that depend on it can be compromised. Therefore, secrets are kept only when necessary.

Analysing how much of the three types of knowledge the attacker knows can give a better understanding of how an attack can be done and how to stop it. In more cases than not, the attackers will have a good knowledge about the learning algorithm, some knowledge about the data, and little knowledge about the feature space [19]. Although it is very unlikely for an attacker to have a *Perfect Knowledge* about the targeted system, it is a common practise to use the perfect knowledge attack to analise the upper bound damage or the worst case of an attack [40].

2.5 The Poisoning Attack

In this project, we will focus on the poisoning attacks, where attackers craft a malicious data and give it to a learning system - trying to make the system learn a wrong model. Every system that trains its classifier with external data are prone to this kind of attack. As concrete examples, these systems include a learning spam filter, a learning malware detector, and probably future wareable devices and driver-less cars. The poisoning attack is therefore a topic that is increasingly important and very much worth exploring. This section will summarise some interesting information about the poisoning attacks that I found during my research.

2.5.1 Attacker's Goal as an Objective Function

A work has been done by Mei and Zhu [28] to formalise the training set attack. They have modeled the *objective* of an attacker O_A in terms of *risk* R_A and *effort* E_A. As such, the formula for an attacker's objective is the following:

$$O_A(D, \hat{\boldsymbol{\theta}}_D) = R_A(\hat{\boldsymbol{\theta}}_D) + E_A(D, D_0) \tag{2.5}$$

where D is the (poisoned) training data, D_0 is the original training data.

Risk

The attacker's risk function R_A is defined as a norm function:

$$R_A(\hat{\boldsymbol{\theta}}_D) = \left\| \hat{\boldsymbol{\theta}}_D - \boldsymbol{\theta}^* \right\| \tag{2.6}$$

in other words, how far away from the *target theta*, $\boldsymbol{\theta}^*$, is the *trained theta*, $\hat{\boldsymbol{\theta}}_D$. Target theta is the set of parameter the attacker wants the machine to learn. Trained theta is the set of parameter the machine actually learn. Therefore the attacker would want the risk value to be as small as possible.

Effort

The attacker's effort function E_A is defined as a norm function:

$$E_A(D, D_0) = \|\boldsymbol{X} - \boldsymbol{X}_0\| \tag{2.7}$$

where $\boldsymbol{X}$ is the design matrix of a training data set D; i.e. matrix that contains all features of all data points in the training set, and $\boldsymbol{X}_0$ is the design matrix of the original training data set D_0. It describes how much alteration the attacker has made to the training data. In many cases, the attacker is restricted to altering a limited portion of the training data. In other cases, altering the training data comes with a cost. Therefore, a successful attack that require less effort is considered a more optimal one. To think about this at a higher level, for two attacks that cause equal damage to the targeted system, it only make sense that the one that require less alteration in the data is more superior. In addition, poisoning points that require less effort would be less likely to be caught by anomaly detectors.

Attacker's Optimisation Problem

With this objective function, the goal of the attacker can be formulate as follow:

$$\begin{aligned} \underset{D \in \mathbb{D}, \hat{\boldsymbol{\theta}}_D}{\text{minimise}} \quad & O_A(D, \hat{\boldsymbol{\theta}}_D) \\ \text{subject to} \quad & \hat{\boldsymbol{\theta}}_D \in \underset{\boldsymbol{\theta} \in \Theta}{argmin} O_L(D, \boldsymbol{\theta}) \end{aligned} \tag{2.8}$$

where O_L an objective function of the learner, and again $\hat{\boldsymbol{\theta}}_D$ is the parameter learned from the data set D. $\mathbb{D}$ here represent the space of D i.e. set of all possible alteration of D_0 that the attacker can make.

In the equation 2.8 shown above where O_A essentially measures the difference between $\boldsymbol{\theta}^*$ and $\hat{\boldsymbol{\theta}}_D$, we can see that given a target model $\boldsymbol{\theta}^*$, one can formulate and solve for the optimal attacking data set D which can be used to *teach* the learning machine a target model $\boldsymbol{\theta}^*$ he wants. This concept of attack proposed by Mei and Zhu is called *Machine Teaching*.

2.5.2 Solving the Optimisation Problem

To solve for the optimal attack points is to solve a very difficult problem [31], and this can be expected from the optimisation problem in the equation 2.8 we had above. In order to solve the minimisation problem for the attacker's objective O_A we have to first solve another optimisation for the learner's objective O_L. This property makes this problem a *bi-level* optimisation problem which is known to be NP-hard.

By making these assumptions: attack space is differentiable, and learner has a convex and regular objective O_L, Mei and Zhu have proposed a way to simplify and solve the bi-level optimisation problem using the gradient descent method with the Karush-Khun-Tucker (KTT) conditions [7]. This method of using the KTT condition to solve the bi-level optimisation will be explored further in this project.

2.5.3 Determining the target model for Machine Teaching

The formula 2.8 requires us to know the target model (parameter) $\boldsymbol{\theta}^*$ that we want the system to learn first. However, this could be difficult to obtain. Take an example of an attacker who want to get the learning machine to learn to maximise the misclassification of its inputs, the values of the target model $\boldsymbol{\theta}^*$ is not obvious. Work of Biggio et al. demonstrate how to carry out effective an poisoning attack without knowing the target parameter in advance [45]. This work uses the similar method of assuming the KTT condition and using gradient descent to find the optimal attack point.

2.5.4 Standard Method from Biggio et al.

The work of Biggio et al. [45] shows a way to solve the bi-level optimisation problem in the context of finding the optimal poisoning point. In his work he solved for the poisoning point targeting machine learning feature selection algorithms; LASSO, ridge regression, and elastic net. The following steps cover parts of Biggio's work that describe how to formulate the algorithm to find the optimal poisoning point.

Learner's and Attacker's Objective Function

The equation below shows the feature selection systems that were targeted in the work of Biggio et al.

$$\underset{\mathbf{w},\mathbf{b}}{\text{minimise}} \quad L = \frac{1}{n}\sum_{i=1}^{n} l(y_i, f(\mathbf{x}_i)) + \lambda\Omega(\mathbf{w}) \tag{2.9}$$

where

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$

and

$$l(y, f(\mathbf{x})) = \frac{1}{2}(f(\mathbf{x}) - y)^2$$

where $\mathbf{w}$ and b are the weights (for input and bias respectively) of the optimal system, $\mathbf{x}$ and y are training sample and its label respectively.

It was pointed out that all three systems – LASSO, ridge-regression, and elastic net – use quadratic loss function, hence the quadratic formula for $l(y, f(\mathbf{x}))$. The term $\Omega(\mathbf{w})$ is the regularisation term. They are different for each feature selection system. For example, LASSO uses l-1 regularisation, therefore $\Omega(\mathbf{w}) = \sum_{i=1}^{n} |w_i|$ Further details on how this works do not play a part in the solving for the optimal attack point.

The attackers' goal can now be formulated as follow:

$$\begin{aligned} \underset{\mathbf{x}_c}{\text{maximise}} \quad & W = \frac{1}{m}\sum_{j=1}^{m} l(\hat{y}_j, f(\hat{\mathbf{x}}_j)) + \lambda\Omega(\mathbf{w}) \\ \text{subject to} \quad & \mathbf{w}, b = \underset{\mathbf{w},b}{argmin} L(\bar{D} \cup \{\mathbf{x}_c\}) \end{aligned} \tag{2.10}$$

where $\hat{x}$ and $\hat{y}$ are a sample from the validation set and its label, and $\bar{D}$ is the training dataset that the attacker assumes the learning system has.

The formula shows an optimisation problem to find the poisoning point $\mathbf{x}_c$ that maximises the cost for the learning systems. Notice the weights $\mathbf{w}$ and b in Equation 2.10 are the outputs of the inner optimisation problem L. As discussed in Section 4.1.1, modifying $\mathbf{x}_c$ will cause $\mathbf{w}$ and b to change. In Biggio's work, he has assumed that the attacker does not know the full training dataset that the targeted system will use when it trains for optimal w and b, so he has used $\bar{D}$ to represent the training data that the attackers have acquired (and assumed that the targeted machine will use). In my work, $\bar{D} = D$ since attackers are allowed full knowledge about the training dataset.

Finding Gradient

In order to avoid searching blindly for the optimal point, attackers could use the gradient descent method to estimate it. To do this, they would have to compute the gradient of their objective function, in this case, W against the value of the poisoning point $\mathbf{x}_c$ i.e. the value $\frac{\delta W}{\delta \mathbf{x}_c}$. In practice, they would have to formulate $\frac{\delta W}{\delta \mathbf{x}_c}$ as a formula of $\mathbf{x}_c$. The following steps show how the term is formulated.

From Equation 2.10:

$$\begin{aligned} W &= \frac{1}{m}\sum_{j=1}^{m} l(\hat{y}_j, f(\hat{\mathbf{x}}_j)) + \lambda\Omega(\mathbf{w}) \\ &= \frac{1}{m}\sum_{j=1}^{m} \frac{1}{2}(f(\hat{\mathbf{x}}_j) - \hat{y}_j)^2 + \lambda\Omega(\mathbf{w}) \end{aligned} \tag{2.11}$$

Using Chain Rule, $\frac{\delta W}{\delta \mathbf{x}_c} = \frac{\delta W}{\delta \mathbf{w}} \cdot \frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}$:

$$\begin{aligned} \frac{\delta W}{\delta \mathbf{x}_c} &= \frac{1}{m}\sum_{j=1}^{m} \frac{1}{2}\frac{\delta}{\delta \mathbf{x}_c}(f(\hat{\mathbf{x}}_j) - \hat{y}_j)^2 + \lambda\frac{\delta}{\delta \mathbf{x}_c}\Omega(\mathbf{w}) \\ &= \frac{1}{m}\sum_{j=1}^{m}(f(\hat{\mathbf{x}}_j) - \hat{y}_j)\frac{\delta}{\delta \mathbf{x}_c}(f(\hat{\mathbf{x}}_j) - \hat{y}_j) + \lambda\frac{\delta\Omega}{\delta \mathbf{w}}\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c} \\ &= \frac{1}{m}\sum_{j=1}^{m}(f(\hat{\mathbf{x}}_j) - \hat{y}_j)\frac{\delta}{\delta \mathbf{x}_c}(\mathbf{w}^T\hat{\mathbf{x}}_j + b - \hat{y}_j) + \lambda\frac{\delta\Omega}{\delta \mathbf{w}}\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c} \\ &= \frac{1}{m}\sum_{j=1}^{m}(f(\hat{\mathbf{x}}_j) - \hat{y}_j)(\hat{\mathbf{x}}_j^T\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c} + \frac{\delta b}{\delta \mathbf{x}_c}) + \lambda\frac{\delta\Omega}{\delta \mathbf{w}}\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c} \end{aligned} \tag{2.12}$$

This formula presented two main problems. First, we do not know $\frac{\delta W}{\delta \mathbf{x}_c}$ and $\frac{\delta b}{\delta \mathbf{x}_c}$. Second, the term $\frac{\delta\Omega}{\delta \mathbf{w}}$ is not unique. The reason for this came from the regularisation term Ω in each feature selection system. In my work, the regularisation term is not considered in the cost function and therefore this second problem will not arise. To overcome the first problem, Biggio et al. made use of the KKT condition of the inner-optimisation problem.

Solving the Problem Using the KKT Conditions

KKT stability conditions for the learning system L states that at the optimal $\mathbf{x}_c$ the objective function L is stable i.e. the gradient of C_{tr} with respect to $\mathbf{w}$ and b is zero, i.e. $\frac{\delta L}{\delta \mathbf{w}} = \mathbf{0}$, and $\frac{\delta L}{\delta b} = 0$.

$$(\frac{\delta L}{\delta \mathbf{w}})^T = \frac{1}{n}\sum_{i=1}^{n}(f(\hat{\mathbf{x}}_i) - \hat{y}_i)\hat{\mathbf{x}}_i + \lambda(\frac{\delta\Omega}{\delta \mathbf{w}})^T = \mathbf{0} \tag{2.13}$$

$$\frac{\delta L}{\delta b} = \frac{1}{n}\sum_{i=1}^{n}(f(\hat{\mathbf{x}}_i) - \hat{y}_i) = 0 \tag{2.14}$$

He then made the assumption that "*the KKT conditions under pertubation of the attack point $\boldsymbol{x}_c$ remains satisfied*". Since the learning system try to optimise L for any input value of $\mathbf{x}_c$, it is sensible to assume that $\frac{\delta L}{\delta \mathbf{w}}$ and $\frac{\delta L}{\delta b}$ remain equals to zero after a small alteration in the value of $\mathbf{x}_c$. Having made that assumption, the followings are true:

$$\frac{\delta}{\delta \mathbf{x}_c}(\frac{\delta L}{\delta \mathbf{w}})^T = \mathbf{0}$$

$$\frac{\delta}{\delta \mathbf{x}_c}(\frac{\delta L}{\delta b}) = 0$$

By differentiating $\frac{\delta L}{\delta \mathbf{w}}$ and $\frac{\delta L}{\delta b}$ with respect to the vector $\mathbf{x}_c$, and letting it equals to zero – following the assumption about the KKT condition. The following linear system can be formed.

$$\begin{bmatrix} \Sigma + \lambda v & \mu \\ \mu^T & 1 \end{bmatrix} \begin{bmatrix} \frac{\delta w}{\delta \mathbf{x}_c} \\ \frac{\delta b}{\delta \mathbf{x}_c}^T \end{bmatrix} = \begin{bmatrix} M \\ \mathbf{w}^T \end{bmatrix} \tag{2.15}$$

where

$$\Sigma = \frac{1}{n} \sum_{i=1}^{n} \hat{\mathbf{x}}_i \hat{\mathbf{x}}_i^T$$

$$\mu = \frac{1}{n} \sum_{i=1}^{n} \hat{\mathbf{x}}_i$$

$$M = \mathbf{x}_c \mathbf{w}^T + (f(\mathbf{x}_c) - y_c) I$$

where I is the Identity Matrix.

Solving the linear system, one could obtain $\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}$ and $\frac{\delta b}{\delta \mathbf{x}_c}$, and by putting them into the equation 2.11, he would be able to perform gradient descent on the optimisation problem W. Note that this linear system has dimensions of $\mathbf{A}_{(d+1)\times(d+1)} * \mathbf{B}_{(d+1)\times d} = \mathbf{C}_{(d+1)\times d}$ where d the number of features in a data point i.e. the dimension of vector $\mathbf{w}$ and $\mathbf{x}_c$.

2.5.5 Counter Measures

This section summarises some state-of-the-art defenses against the causative attacks. For this project, it is important to study them because if we can derive an attack that is able evade these defense mechanisms, or improving the defense strategies, we would likely be contributing to the state-of-the-art.

RONI Defenes

The Reject On Negative Impact defense [19] is a data sanitization technique. As the name suggests, this defense will measure an effect of training with each training sample on a classifier and reject training samples that cause a negative impact to the classifier. To execute this defense, the classifier is trained first with a *base training set* D_b. The trained classifier's performance would be tested with a *quiz* data set. Then upon each new training sample Q, the classifier would be retrain with the training set $D_b \cup \{Q\}$. The performance of this newly trained classifier will be measured with the same *quiz*, and if the performance declined over a certain threshold, this new sample Q will be removed from the training data set.

Dynamic Threshold Defense

This defense [29] is used against the attack that shifts (translate) the entire hypothesis function of a classifier to a certain direction; for example, the dictionary attack for the SpamBayes spam filter where the attacker brings up the spam score of all emails. This defense adjust the threshold value used in classification to an appropriate value that can be used with the shifted hypothesis function. To execute this, we would divide the entire training set in half: first half F for training, and second half V for validation. We train the classifier with F, and use the trained classifier to classify V. The result of the classification of V can be compared with the actual label of V to calculate the appropriate new threshold value of our classifier.

Chapter 3

Project Setup

3.1 Learning Algorithms

In the work I have mainly experimented with three widely-used machine learning classifiers: ADALINE, Logistic Regression, and Multi-Layered Perceptron. The ADALINE and Logistic Regression classifiers are linear classifiers, and the Multi-Layered Perceptron is a non-linear neural network classifier. All three systems are made in Matlab exclusively for this project to ensure full understanding of the classifiers and their implementations – making sure that we have the full-knowledge of the classifiers in order to carry out an optimal poisoning attack.

3.2 Datasets

This section describes the different datasets used for evaluation of the experiments carried out in this project. It is a common practise to split a dataset into 3 sets:

- **Training Set**: the samples used for training the learning systems.
- **Validation Set**: the samples used for adjusting the parameters. In our case, they are the samples used to adjust values in the poisoning point.
- **Test set**: the samples used to evaluate the final performance of the learning system.

All this is done to provide the test performance result that is as fair and realistic as possible. For example, if I had tested the final state of a learning system with the validation set or the training set, the performance will be better than in reality because, during training the learning system have already seen them and used them to optimised its model. Therefore, in practice, we use the test set, which is not seen by the learning system before, to model the real-world data and test for the performance of the learning system.

3.2.1 The Synthetic Dataset

This is a dataset that is created by Gaussian Distribution [27] in the beginning of this project with the aim to use it to do quick checks whether my code was working correctly. The dataset consists of 40 training samples and 400 validation samples. Each sample has 2 features and corresponds to a binary-class label.

3.2.2 The Real Datasets

The three real datasets I experimented with are Spambase, Ransomware, and MNIST datasets [8] [10] [25]. Each datasets are split into 10 sets, each with 100 training samples and 400 validation samples selected randomly from the whole dataset. The rest of the samples in the dataset are used as the test set in each split.

The experiments were performed in 10 repetitions, each time with a different dataset split. By taking the average of the results of these 10 splits, we reduce possible biases in the results induced by the partition of the data.

3.2.3 Spambase

Spambase [8] is a dataset of features extracted from emails, where each sample can belong in one of the two classes: the positive class (1) being the *Spam* emails, or the negative class (0) being the legitimate *Ham* emails. In this project, this dataset is considered the simplest 'real' dataset, as it has the least number of features – 54 binary features. The total number of the dataset is 4601 samples, giving us the test dataset in each of the 10 splits of 4101 samples.

3.2.4 Ransomware

Ransomware dataset is a dataset of ransomwares collected in the work of Daniele Sgandurra et al. [10]. It has 400 binary features, and each sample has a binary label where positive class being a ransomware class. The total number of samples in the dataset is 1079 and therefore each of the 10 splits would have 579 test samples.

3.2.5 MNIST

MNIST [25] is a dataset where each sample is an 28x28 pixel image of a handwritten number digit. It has 784 continuous features ranging from 0 to 1 inclusively, each one representing the intensity value of each pixel. With 784 features, it is the highest-featured dataset I have experimented with in this project.

This dataset has a multi-class label of 10 classes representing each digit. In most experiments in this work, we would have to compare the performance of a poisoning attack method across all datasets. In such situations, to make the dataset comparable to the binary-class datasets, I would modify the label of the MNIST dataset by only filter out digit 1 and 7 from then dataset, creating a binary-class dataset with positive class being digit 7 and negative class being digit 1. The total number of samples of class 1 and 7 in the dataset are 2197, and thus each split of the MNIST binary-class dataset has 1697 test samples.

3.2.6 IRIS

The IRIS dataset [34] is a small dataset of 150 samples each with 4 features and 3 target classes i.e. 3 different labels, representing 3 different classes of iris plants. I have used it to run quick test on the correctness of multi-class code implementations.

3.3 Evaluating the Classifier

3.3.1 Confusion Matrix

To evaluate the performance of a classifier, the confusion matrix is often used as it provides sufficient useful information to evaluate the performance of the classifier. The value in each cell is the true positive, true negative, false positive, and false negative count of all the testing samples classified by a classifier. Later when we examine the results of a multi-class classifier, we would see that the confusion matrix is extended by dividing each value of the cell with the total number of samples in its actual class so that each cell represents the classification (for the diagonal cells) or mis-classification rates (for the non-diagonal cells) of each class.

	actual class 1	actual class 0
predicted class 1	True Positive (TP)	False Positive (FP)
predicted class 0	False Negative (FN)	True Negative (TN)

Table 3.1: Binary-class confusion matrix, where class 1 is the positive class and class 0 is the negative class

3.3.2 Classification Rate and Classification Error

$$Classification\ Rate = \frac{TP}{TP + TN + FP + FN}$$

and

$$Classification\ Error = 1 - Classification\ Rate$$

These two values are used to summarise the performance of a classifier on its classification task.

3.3.3 False Positive Rate and False Negative Rate

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

$$False\ Negative\ Rate = \frac{FN}{FN + TP}$$

The False Positive and False Negative rates are the probabilities that the classifier mis-classify a sample from the negative and positive class respectively. These values are useful for the targeted attack analysis.

3.4 Specific Problems

3.4.1 Value of Sample Labels

In our dataset, the default label for the positive and negative classes are the value 1 and 0 respectively. When training a binary-class ADALINE classifier, using the label as 1 and -1 for the positive and negative class will result in a better classification performance of the classifier. This is because the output function of the ADALINE i.e. $o = \mathbf{w}^T\mathbf{x}$ could produce negative output. The classifier would then be able to separate the two class easier when the label is allowed to be negative. This is different for the case of the Logistic Regression classifier where the output function is a sigmoid function $o = \sigma(\mathbf{w}^T\mathbf{x})$. The output would strictly be in the range of 0 and 1, and thus making the class labels of 0 and 1 suitable for the Logistic Regression Classifier.

3.4.2 Finding Reasonable Parameters Values

One of the problems this project faces is the fact that each of my proposed methods are based on the gradient descent which requires the user to input a suitable set of parameters; specifically, the learning rates and the number of training iterations. By inputing different values of the parameters, the overall performance of a classifier could be drastically affected. In the example of training a machine learning classifier with two different datasets, finding the optimal values of the parameters means finding the parameters which results in a system that produce minimal classification error. This is problem is similar to the problem of finding the optimal hyper-parameters, which is a difficult problem [6] [5]. Furthermore, the optimal parameter values of a classifier when solving one dataset problem is also very likely to be different from when solving another dataset problem.

It is worth pointing out that for a simple convex problem, we could use any small value of learning rates and high number of iterations to train the system as it will never get stuck in the local minima. In this work, however, I will be solving a bi-level optimisation problem. Such problems is not convex and thus finding the optimal parameters remains a difficult problem.

To get a working solution, in most cases, we are not required to find '*the optimal*' parameters i.e. a good estimation would suffice. In this project, I looked for those parameters values by trial-and-error, using the cost against iteration graph to guide my estimation. A working solution is one with smooth increasing or decreasing cost graph such as in figures 3.1 and 3.2.

3.4.3 Initialisation of Poisoning Point

The problem of non-convexity of our problem also poses difficulty in choosing an initial point for our gradient descent based method. Different initial points could result in different overall performance as the point could lead to different local minima.

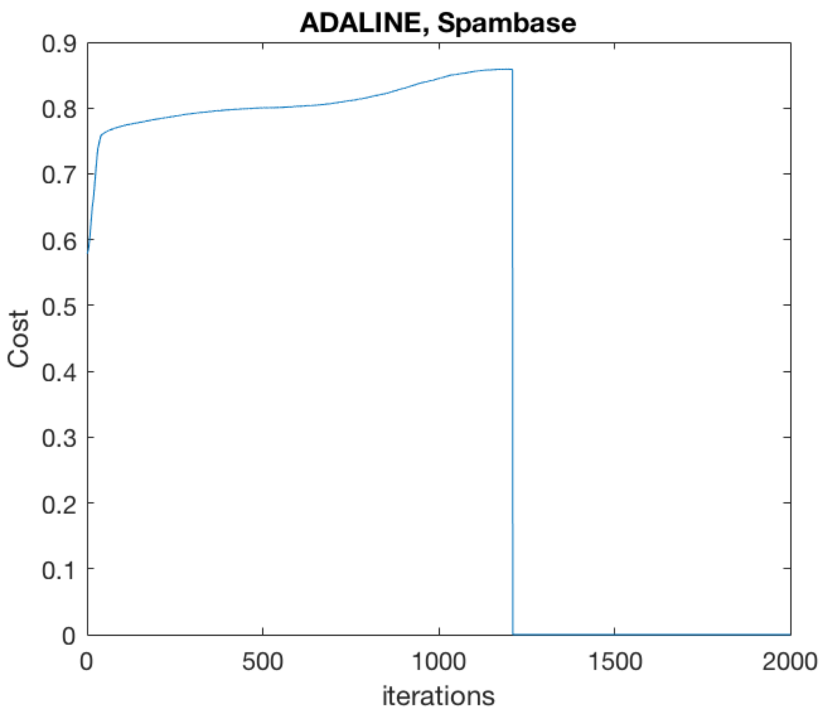


Figure 3.1: Graph of cost against number of update iterations for outer-optimisation (maximisation) problem. The updating algorithm was stopped after about 1200 iterations by the *early-stopping* mechanism to avoid unnecessary computation

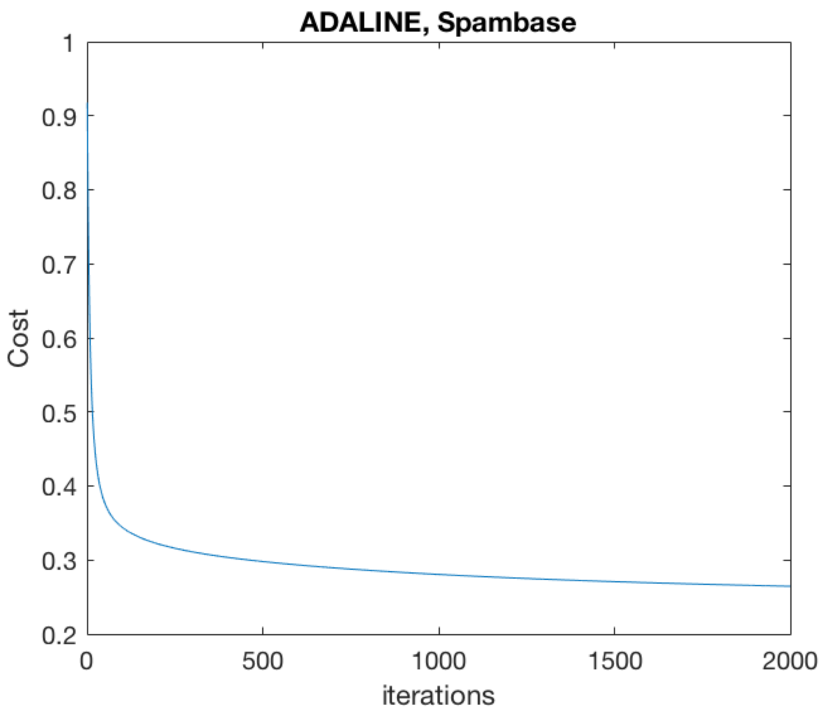


Figure 3.2: Graph of cost against number of update iterations for the inner-optimisation (minimisation) problem.

Chapter 4

Optimal Poisoning Point & the Direct Method

4.1 Optimal Poisoning attacks

This work focuses on the Poisoning Attack on Machine Learning Algorithms. In this type of attack, attackers have control over a portion of the targeted system's training data. They would also have a certain amount of knowledge about the training algorithm of the targeted system. In this work, in order to examine the robustness of the learning algorithm against such attack, I have analysed the extreme cases – ones where the attackers have full knowledge about the training data as well as the training algorithm. With this knowledge, a crucial task still remains for the attackers to find the most effective training samples which, when injected into the training dataset, would cause most damage to the targeted system – causing highest rise in the error rate.

This work solved a slightly different optimisation problem from what Mei and Zhu [28] have proposed i.e. equation 2.5. The problem described in the Background Section tries to look for a 'working' *poisoning data point*, that requires least tampering of the legitimate training dataset. However, for this project, it would be more useful to find the 'most damaging' point and observe how different machine learning algorithms are affected by it as shown in the work of Biggio et al. [45]. Thus, we optimise on the poisoning point according to the following equation.

$$\begin{aligned} \underset{\mathbf{x}_c \in D}{\text{maximise}} \quad & C_{val}(\mathbf{w}, b) \\ \text{subject to} \quad & \mathbf{w}, b = \underset{\mathbf{w},b}{argmin} C_{tr}(D, \mathbf{w}, b) \end{aligned} \tag{4.1}$$

These two problems are both bi-level optimisation problems, therefore the same techniques that I have used in this work could also be used with the problem proposed by Mei and Zhu.

The following sections will explain how to solve the optimisation problem using gradient descent. Specifically, how to find the gradient to use for the gradient descent $\frac{\delta C_{val}}{\delta \mathbf{x}_c}$. This chapter will discuss the standard (direct) way of computing that gradient, and will show and discuss why it is not the most efficient and stable method.

4.1.1 Bi-Level Optimisation problem Difficulty

I would like to emphasise on the difficulty of solving a bi-level optimisation problem. It is already difficult to solve a single-level optimisation problem when the problem has many constraints i.e. have high dimension, as we cannot perform a search algorithm to find the optimal solution as we could with a simple, low-dimensional optimisation problem. In our case, the solution for the inner-optimisation problem can be found if the training system's cost against weights graph for that problem is *convex*, which is only the case for the linear classifiers. If a optimisation problem is convex, we can solve it well with gradient descent. However, if it is non-convex, the best we can do is to estimate the optimal solution with gradient descent.

Solving bi-level optimisation problem is therefore exponentially more difficult than solving a single-level optimisation problem, as when searching for the optimal solution $\mathbf{x}_c$ for the outer-problem, the optimal solution for the inner-problem, in our case $\mathbf{w}$ and b, has to be re-computed each time the value of $\mathbf{x}_c$ changes. Later in this report, we will see how to solve this bi-level optimisation problem efficiently.

4.1.2 Attacker's Capability

Throughout this work, with an exception of the indiscriminate multi-class classifier attack, I have made the assumption that the attacker can only generate poisoning points in a positive class i.e. all poisoning points that the attacker can generate has to be labeled as positive-class points. This is a very reasonable assumption for problems like spam email classification in a machine learning spam-filtering application [29]. This reason for this is that the spam-filtering application user decides whether an email is of a positive(spam) class or not, therefore it is very unlikely that the attacker will be able to generate enough malicious-legitimate emails to create a substantial impact with the poisoning attack. The attacker, on the other hand, can send as much spam emails as he likes, and these positive-class samples will eventually get into the training dataset when the spam-filtering application retrains its classification system.

To be consistent in all problems, I have made this restriction for both ransomware and MNIST problem as well. The same argument about the attacker's capability could be applied to a ransomware detection system as the spam-filtering application. It is also worth noting that by restricting the attacker's capability as such, the experiments in this work would be also consistent with the literature.

4.1.3 Black-Box Characteristic

Normally, to craft a poisoning sample, an attacker has to study the dataset that he is trying to poison. For example, in the MNIST problem, he would have to know that the digit 1 looks similar to digit 7, and try to create a poisoning sample with such knowledge of the dataset.

By simply solving the attacker's bi-level optimisation without doing a deeper analysis for each dataset, I have created another layer of abstraction for the attackers. The methods described in this project report will be able to estimate the optimal poisoning sample from any dataset. This means that the attacker no longer have to study the dataset in order to craft a poisoning sample.

4.2 The Standard Method

This section explains how Biggio's method [45], as described in section 2.5.4, is adapted to work for the optimisation problem proposed in this project. This project first look at the two machine learning classifier system – ADALINE and Logistic Regression – and examine how sensitive to the poisoning attack they are. This standard method adapted from Biggio's work [45] can be used to craft the optimal poisoning points for both classifiers.

In this work, I will describe the attacker's goal as:

$$\underset{\mathbf{x}_c}{\text{maximise}} \quad C_{val} = \frac{1}{n}\sum_{i=1}^{n} l(\hat{y}_i, f(\hat{\mathbf{x}}_i, \mathbf{w})) \tag{4.2}$$

where C_{val} is the *cost* (output of the cost function) calculated using the validation dataset, $\hat{\mathbf{x}}$ and $\hat{y}$ are the validation set data points and their labels, $f(\mathbf{x}, \mathbf{w})$ is the hypothesis function, which depended on weights $\mathbf{w}$ and b – the results of the learner's optimisation problem, equation 4.3.

$$\underset{\mathbf{w},\mathbf{b}}{\text{minimise}} \quad C_{tr} = \frac{1}{n}\sum_{i=1}^{n} l(y_i, f(\mathbf{x}_i, \mathbf{w})) \tag{4.3}$$

4.2.1 ADALINE Classifier

For ADALINE, the calculations for finding the gradient $\frac{\delta C_{val}}{\delta \mathbf{x}_c}$ are almost identical to the methods shown in Biggio's work. This is because the loss function l and the hypothesis function $f(\mathbf{x}, \mathbf{w})$ are the same. Therefore, we would have the equations 4.2, and 4.3, with

$$f(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T\mathbf{x} + b$$

and

$$l(y, f(\mathbf{x}, \mathbf{w})) = \frac{1}{2}(f(\mathbf{x}, \mathbf{w}) - y)^2$$

The equation for finding the gradient for the poisoning point gradient descent would then be:

$$\frac{\delta C_{val}}{\delta \mathbf{x}_c} = \frac{1}{m}\sum_{j=1}^{m}(f(\hat{\mathbf{x}}_j) - \hat{y}_j)(\hat{\mathbf{x}}_j^T\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c} + \frac{\delta b}{\delta \mathbf{x}_c}) \tag{4.4}$$

With the assumption that KKT conditions of the targeted system (inner optimisation problem) remain satisfied after small perturbations in $\mathbf{x}_c$, the following equations would be true:

$$\frac{\delta}{\delta \mathbf{x}_c}(\frac{\delta C_{tr}^T}{\delta \mathbf{w}}) = \mathbf{0}_{d\times d} \tag{4.5}$$

$$\frac{\delta}{\delta \mathbf{x}_c}(\frac{\delta C_{tr}}{\delta b}) = \mathbf{0}_{d} \tag{4.6}$$

We have that

$$\frac{\delta C_{tr}^T}{\delta \mathbf{w}} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{w}^T\mathbf{x}_i + b - y_i)\mathbf{x}_i \tag{4.7}$$

and

$$\frac{\delta C_{tr}^T}{\delta b} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{w}^T\mathbf{x}_i + b - y_i) \tag{4.8}$$

and,

$$\begin{aligned}
\frac{\delta}{\delta \mathbf{x}_c}(\frac{\delta C_{tr}^T}{\delta \mathbf{w}}) &= \frac{1}{n}\sum_{i=1}^{n}[(\mathbf{w}^T\mathbf{x}_i + b - y_i)\frac{\delta \mathbf{x}_i}{\delta \mathbf{x}_c} + \mathbf{x}_i(\frac{\delta}{\delta \mathbf{x}_c}(\mathbf{w}^T\mathbf{x}_i + b - y_i))^T] \\
&= \frac{1}{n}[(\mathbf{w}^T\mathbf{x}_c + b - y_c)\mathbb{I}_{d\times d} + (\sum_{i\neq c}\mathbf{x}_i(\frac{\delta}{\delta \mathbf{x}_c}(\mathbf{w}^T\mathbf{x}_i + b - y_i))^T) \\
&\quad + \mathbf{x}_c(\frac{\delta}{\delta \mathbf{x}_c}(\mathbf{w}^T\mathbf{x}_c + b - y_c))^T] \\
&= \frac{1}{n}[(\sum_{i\neq c}\mathbf{x}_i(\frac{\delta \mathbf{w}^T}{\delta \mathbf{x}_c}\mathbf{x}_i + \frac{\delta b}{\delta \mathbf{x}_c})^T) + \mathbf{x}_c(\frac{\delta\ \mathbf{w}^T}{\delta \mathbf{x}_c}\mathbf{x}_c + \mathbf{w} + \frac{\delta b}{\delta \mathbf{x}_c})^T \\
&\quad + (\mathbf{w}^T\mathbf{x}_c + b - y_c)\mathbb{I}_{d\times d}] \\
&= \frac{1}{n}[(\sum_{i\neq c}\mathbf{x}_i(\mathbf{x}_i^T\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}) + (\frac{\delta b}{\delta \mathbf{x}_c})^T) + \mathbf{x}_c(\mathbf{x}_c^T\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c} + \mathbf{w}^T + \mathbf{x}_c^T(\frac{\delta b}{\delta \mathbf{x}_c})^T) \\
&\quad + (\mathbf{w}^T\mathbf{x}_c + b - y_c)\mathbb{I}_{d\times d}] \\
&= \frac{1}{n}[(\sum_{i=1}^{n}\mathbf{x}_i(\mathbf{x}_i^T\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}) + (\frac{\delta b}{\delta \mathbf{x}_c})^T) + \mathbf{x}_c\mathbf{w}^T + (\mathbf{w}^T\mathbf{x}_c + b - y_c)\mathbb{I}_{d\times d}]
\end{aligned} \tag{4.9}$$

and

$$\begin{aligned}
\frac{\delta}{\delta \mathbf{x}_c}(\frac{\delta C_{tr}^T}{\delta b}) &= \frac{1}{n}[(\sum_{i\neq c}\frac{\delta}{\delta \mathbf{x}_c}(\mathbf{w}^T\mathbf{x}_i + b - y_i)) + \frac{\delta}{\delta \mathbf{x}_c}(\mathbf{w}^T\mathbf{x}_c + b - y_c)] \\
&= \frac{1}{n}[(\sum_{i\neq c}(\frac{\delta \mathbf{w}^T}{\delta \mathbf{x}_c}\mathbf{x}_i + \frac{\delta b}{\delta \mathbf{x}_c})) + (\frac{\delta \mathbf{w}^T}{\delta \mathbf{x}_c}\mathbf{x}_c + \mathbf{w} + \frac{\delta b}{\delta \mathbf{x}_c})] \\
&= \frac{1}{n}[(\sum_{i=1}^{n}(\frac{\delta \mathbf{w}^T}{\delta \mathbf{x}_c}x_i + \frac{\delta b}{\delta \mathbf{x}_c})) + \mathbf{w}]
\end{aligned} \tag{4.10}$$

Therefore, from equation 4.9, we would have that,

$$\frac{1}{n}[(\sum_{i=1}^{n}\mathbf{x}_i(\mathbf{x}_i^T\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}) + (\frac{\delta b}{\delta \mathbf{x}_c})^T) + \mathbf{x}_c\mathbf{w}^T + (\mathbf{w}^T\mathbf{x}_c + b - y_c)\mathbb{I}_{d\times d}] = \mathbf{0}_{d\times d}$$

$$\frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i\mathbf{x}_i^T(\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}) + \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i(\frac{\delta b}{\delta \mathbf{x}_c})^T = -\frac{1}{n}(\mathbf{x}_c\mathbf{w}^T + (\mathbf{w}^T\mathbf{x}_c + b - y_c)\mathbb{I}_{d\times d}) \tag{4.11}$$

and from 4.10,

$$\frac{1}{n}[(\sum_{i=1}^{n}(\frac{\delta \mathbf{w}^T}{\delta \mathbf{x}_c}x_i + \frac{\delta b}{\delta \mathbf{x}_c})) + \mathbf{w}] = \mathbf{0}_d$$

$$\frac{1}{n}[(\sum_{i=1}^{n}(\mathbf{x}_i^T\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c} + (\frac{\delta b}{\delta \mathbf{x}_c})^T)) + \mathbf{w}^T] = \mathbf{0}_d$$

$$\frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i^T(\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}) + \frac{1}{n}(\frac{\delta b}{\delta \mathbf{x}_c})^T)) = -\frac{1}{n}\mathbf{w}^T \tag{4.12}$$

With the equations 4.11 and 4.12, a similar linear system as in Biggio's work can be formed like so:

$$\begin{bmatrix} \Sigma & \mu \\ \mu^T & \frac{1}{n} \end{bmatrix} \begin{bmatrix} \frac{\delta \mathbf{w}}{\delta \mathbf{x}_c} \\ \frac{\delta \mathbf{b}}{\delta \mathbf{x}_c}^T \end{bmatrix} = -\frac{1}{n}\begin{bmatrix} M \\ \mathbf{w}^T \end{bmatrix} \tag{4.13}$$

where

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i\mathbf{x}_i^T$$

$$\mu = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i$$

$$M = \mathbf{x}_c\mathbf{w}^T + (\mathbf{w}^T\mathbf{x} + b - y_c)\mathbb{I}_{d\times d}$$

where $\mathbb{I}$ is the Identity Matrix.

Solving the system 4.20 allows us to obtain $\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}$ and $\frac{\delta b}{\delta \mathbf{x}_c}$ which then can be substitute into the equation 4.4 to solve for the attacker's optimisation problem gradient $\frac{\delta C_{val}}{\delta \mathbf{x}_c}$.

4.2.2 Logistic Regression Classifier

Similar approach can be used with Logistic Regression. The only differences are the loss function l and the activation function f(x):

$$f(\mathbf{x}, \mathbf{w}) = \sigma(\mathbf{w}^T\mathbf{x} + b)$$

and

$$l(f(\mathbf{x}, \mathbf{w}), y) = y\log(f(\mathbf{x}, \mathbf{w}) + (1-y)\log(1 - f(\mathbf{x}))$$

Thus,

$$C_{val} = \frac{1}{m}\sum_{j=1}^{m} y_j log(\sigma(\mathbf{w}^T\hat{\mathbf{x}}_j + b)) + (1-y_j)log(1 - \sigma(\mathbf{w}^T\hat{\mathbf{x}}_j + b)) \tag{4.14}$$

$$\frac{\delta C_{val}}{\delta \mathbf{x}_c} = \frac{1}{m}\sum_{j=1}^{m}(\sigma(\mathbf{w}^t\hat{\mathbf{x}}_j + b) - y_i)(\hat{\mathbf{x}}_j^T\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c} + \frac{\delta b}{\delta \mathbf{x}_c}) \tag{4.15}$$

KKT condition:

$$\frac{\delta C_{tr}^T}{\delta \mathbf{x}_c} = \frac{1}{n}\sum_{i=1}^{n}(\sigma(\mathbf{w}^t\mathbf{x}_j + b) - y_i)\mathbf{x}_i = \mathbf{0}_d \tag{4.16}$$

$$\frac{\delta C_{tr}^T}{\delta b} = \frac{1}{n}\sum_{i=1}^{n}(\sigma(\mathbf{w}^t\mathbf{x}_j + b) - y_i) = 0 \tag{4.17}$$

The KKT condition of the inner optimisation problem remains satisfied with small perturbation of $\mathbf{x}_c$:

$$\frac{\delta}{\delta \mathbf{x}_c}(\frac{\delta C_{tr}^T}{\delta \mathbf{w}}) = \mathbf{0}_{d\times d} \tag{4.18}$$

$$\frac{\delta}{\delta \mathbf{x}_c}(\frac{\delta C_{tr}}{\delta b}) = \mathbf{0}_d \tag{4.19}$$

By differentiating $\frac{\delta C_{tr}^T}{\delta \mathbf{w}}$ and $\frac{\delta C_{tr}}{\delta b}$ with respect to $\mathbf{x}_c$ and letting them equal to zero as we have done for Adaline, we can obtain a similar linear system:

$$\begin{bmatrix} \Sigma & \mu \\ \mu^T & \alpha \end{bmatrix} \begin{bmatrix} \frac{\delta \mathbf{w}}{\delta \mathbf{x}_c} \\ \frac{\delta b}{\delta \mathbf{x}_c}^T \end{bmatrix} = - \begin{bmatrix} M \\ p \end{bmatrix} \tag{4.20}$$

where

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}((\sigma(\mathbf{w}^T\mathbf{x}_i + b)(1 - \sigma(\mathbf{w}^T\mathbf{x}_i + b)))\mathbf{x}_i\mathbf{x}_i^T$$

$$\mu = \frac{1}{n}\sum_{i=1}^{n}((\sigma(\mathbf{w}^T\mathbf{x}_i + b)(1 - \sigma(\mathbf{w}^T\mathbf{x}_i + b)))\mathbf{x}_i$$

$$\alpha = \frac{1}{n}\sum_{i=1}^{n}((\sigma(\mathbf{w}^T\mathbf{x}_i + b)(1 - \sigma(\mathbf{w}^T\mathbf{x}_i + b)))$$

$$M = \frac{1}{n}(\sigma(\mathbf{w}^T\mathbf{x}_c + b) - y_c)\mathbb{I}_{d\times d} + \frac{1}{n}(\sigma(\mathbf{w}^T\mathbf{x}_c + b)(1 - \sigma(\mathbf{w}^T\mathbf{x}_c + b)))\mathbf{x}_c\mathbf{w}^T$$

$$p = \frac{1}{n}(\sigma(\mathbf{w}^T\mathbf{x}_c + b)(1 - \sigma(\mathbf{w}^T\mathbf{x}_c + b)))\mathbf{x}_c\mathbf{w}^T$$

where $\mathbb{I}$ is the Identity Matrix.

4.2.3 Method's Limitation

This method of finding the gradient will suffer a scalability problems with the numbers of features. This is a result of solving for $\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}$ and $\frac{\delta b}{\delta \mathbf{x}_c}$ in equation 2.15 by computing the matrix inverse as shown.

$$\begin{bmatrix} \frac{\delta \mathbf{w}}{\delta \mathbf{x}_c} \\ \frac{\delta \mathbf{b}}{\delta \mathbf{x}_c}^T \end{bmatrix} = -\frac{1}{n} \begin{bmatrix} \Sigma & \mu \\ \mu^T & 1 \end{bmatrix}^{-1} \begin{bmatrix} M \\ w^T \end{bmatrix} \tag{4.21}$$

The complexity for computing the matrix inverse is known to be $O(di^3)$ where di is the dimension of the square matrix, in this case it is the number of the features in a data point plus one for the bias term, i.e. $di = d + 1$.

To overcome this problem, Conjugate Gradient method could be used to solve the linear system. We will see this in the next chapter.

Chapter 5

Poisoning with Conjugate Gradient Method

5.1 Conjugate Gradient Optimisation

Conjugate Gradient [18] is an iterative method that solves a system of linear equation $\mathbf{Ax} = \mathbf{b}$ without calculating the matrix inverse $\mathbf{A}^{-1}$, avoiding its $O(di^3)$ complexity, thus improving the speed performance of the optimisation problem drastically. By avoiding the computation of the matrix inverse, the conjugate-gradient method is more stable and more efficient than the standard method. The complexity of Conjugate Gradient algorithm is equals to that of $\mathbf{Ax}$.

5.1.1 Conjugate Gradient Algorithm

Algorithm 1 Conjugate Gradient Algorithm [43]

```
Require: matrix $\mathbf{A}$, and vector $\mathbf{b}$
 1: $\mathbf{x}_0 \leftarrow \mathbf{0}$
 2: $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$
 3: $k \leftarrow 0$
 4: while $true$ do
 5:     $\alpha_k \leftarrow (\mathbf{r}_k^T \mathbf{r}_k)/(\mathbf{p}_k^T A \mathbf{p}_k)$
 6:     $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$
 7:     $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k + \alpha_k \mathbf{A}\mathbf{p}_k$
 8:     if $\mathbf{r}_{k+1}$ is sufficiently small then
 9:         exit loop
10:     end
11:     $\beta_k \leftarrow (\mathbf{r}_{k+1}^T \mathbf{r}_{k+1})/(\mathbf{r}_k^T \mathbf{r}_k)$
12:     $\mathbf{p}_{k+1} \leftarrow \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
13:     $k \leftarrow k + 1$
14: end
15: return $\mathbf{x}_{k+1}$
```

With this method, one could solve the linear system for $\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}$ and $\frac{\delta b}{\delta \mathbf{x}_c}$ in equation 2.15 without having to compute the matrix inverse. Notice that the equation 2.15

is not a matrix-vector multiplication as proposed in the Conjugate Gradient algorithm, instead it is a matrix-matrix multiplication. However, this is not a problem as solving a matrix-matrix multiplication system is the same as solving matrix-vector for each column of the second matrix. Recall that the system in equation 2.15 has the following dimensions $\mathbf{A}_{(d+1)\times(d+1)} * \mathbf{B}_{(d+1)\times d} = \mathbf{C}_{(d+1)\times d}$ where d is the number of features in a data point. This means that solving for the matrix $\mathbf{B}$ would require applying the conjugate gradient algorithm d times.

To solve the attacker's optimisation problem with gradient descent the gradient of the outer optimisation problem could then be computed directly by substituting $\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}$ and $\frac{\delta b}{\delta \mathbf{x}_c}$ into the equation of this form

$$\frac{\delta C_{val}}{\delta \mathbf{x}_c} = \frac{\delta C_{val}}{\delta \mathbf{w}} \cdot \frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}$$

where $\frac{\delta C_{val}}{\delta \mathbf{w}}$ is directly differentiated. Examples of such in equations are equation 2.12, 4.4, and 4.15

5.1.2 Improved Method

As mentioned in the previous section, using conjugate gradient to solve the matrix-matrix multiplication system $\mathbf{A}_{(d+1)\times(d+1)} * \mathbf{B}_{(d+1)\times d} = \mathbf{C}_{(d+1)\times d}$ would result in having to perform conjugate gradient d times. To solve this more efficiently, I studied the work by Andrew Ng et al. [9] and was able to rearrange the problem to reduce the number of times conjugate gradient is performed – in order to find $\frac{\delta w}{\delta x_c}$ and $\frac{\delta b}{\delta x_c}$, i.e. $\mathbf{B}_{(n+1)\times(n)}$ – from d-times to only one time. This is a significant improvement in performance as the number of features (d) could be very large in some datasets. A data point in MNIST dataset, for example, has 784 features.

This improved method makes use of the *Implicit Function Theorem* [22] and the *Finite Difference Method* [33].

Finite Difference

The finite difference *trick* is also described in Andrew Ng's work [9]. It allows us to estimate the product of a second-order derivative and a vector efficiently, especially in the high-dimensional problem where we look for the second-order derivative with respect to vectors. The equation for finite differencing is as follow:

$$\frac{\delta}{\delta \mathbf{x}}(\frac{\delta}{\delta y} f(\mathbf{x})) \cdot \mathbf{v} = \lim_{r \to 0} \frac{\frac{\delta}{\delta y} f(\mathbf{x} + r\mathbf{v}) - \frac{\delta}{\delta y} f(\mathbf{x})}{r} \tag{5.1}$$

where $\mathbf{v}$ can be any arbitrary vector. It is worth noticing the resemblance of this equation with the first principle of differentiation.

Implicit Function

The implicit function theorem allows us to differentiate a function $g(x)$ with respect to a variable y that is not explicitly defined in the function, by differentiating another function $f(g, x)$ provided that f is a continuously differentiable function.

In our case, we would like to know the derivative $\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c}$, and since $w = \underset{\mathbf{w}}{argmin} L(\hat{D} \cup \{\mathbf{x}_c\})$, $\mathbf{w}$ is not explicitly defined by $\mathbf{x}_c$ but we know that it depends on it. Therefore we can apply the implicit function theorem we would give us the following equation:

$$\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c} = (\frac{\delta f}{\delta \mathbf{w}})^{-1} \cdot \frac{\delta f}{\delta \mathbf{x}_c} \tag{5.2}$$

where we let $f(\mathbf{w}, \mathbf{x}_c) = \frac{\delta}{\delta \mathbf{w}} C_{tr}(\mathbf{w}, \mathbf{x}_c)$, and we know that f is a continuously differentiable function because from the KKT condition we know that f is smooth.

Method's Proof

This subsection covers the proof of procedure taken to compute the attacker's optimisation gradient $\frac{\delta C_{val}}{\delta \mathbf{x}_c}$ with the new improved method.

1. The aim is to find $\frac{\delta C_{val}}{\delta \mathbf{x}_c}$. Performing the chain rule, we would get the formula below:

$$\frac{\delta C_{val}}{\delta \mathbf{x}_c} = ((\frac{\delta C_{val}}{\delta \mathbf{w}})^T \cdot \frac{\delta \mathbf{w}}{\delta \mathbf{x}_c})^T$$

2. Re-arrange the $\frac{\delta C_{val}}{\delta x_c}$ equation to get:

$$\frac{\delta C_{val}}{\delta \mathbf{x}_c} = (\frac{\delta \mathbf{w}}{\delta \mathbf{x}_c})^T \cdot \frac{\delta C_{val}}{\delta \mathbf{w}} \tag{5.3}$$

3. By KKT-condition of the inner optimisation problem, and implicit function:

$$\frac{\delta C_{val}}{\delta \mathbf{x}_c} = -(\frac{\delta^2 C_{tr}}{\delta \mathbf{x}_c \delta \mathbf{w}})(\frac{\delta^2 C_{tr}}{\delta \mathbf{w}^2})^{-1} \cdot \frac{\delta C_{val}}{\delta \mathbf{w}} \tag{5.4}$$

4. Taking the second and third term of the right hand side of the equation 5.4 let that equals to vector $\mathbf{z}$:

$$\frac{\delta C_{val}}{\delta \mathbf{x}_c} = -(\frac{\delta^2 C_{tr}}{\delta \mathbf{x}_c \delta \mathbf{w}})\mathbf{z} \tag{5.5}$$

and

$$\mathbf{z} = (\frac{\delta^2 C_{tr}}{\delta \mathbf{w}^2})^{-1} \cdot \frac{\delta C_{val}}{\delta \mathbf{w}} \tag{5.6}$$

The equation could be re-arrange in the form $\mathbf{Az} = \mathbf{b}$:

$$(\frac{\delta^2 C_{tr}}{\delta \mathbf{w}^2})\mathbf{z} = \frac{\delta C_{val}}{\delta \mathbf{w}}$$

where $\mathbf{z}$ can be solved using the conjugate gradient algorithm. It is worth pointing out that, within this conjugate gradient computation, the matrix-vector product $(\frac{\delta^2 C_{tr}}{\delta \mathbf{w}^2})\mathbf{z}$ has to be computed multiple times to find the 'right' value of $\mathbf{z}$. This product can be computed with the finite difference method.

5. Having obtained the vector $\mathbf{z}$ in the previous step, we can now solve the equation 5.5 for $\frac{\delta C_{val}}{\delta x_c}$ using again the finite difference 9.2 method:

$$\frac{\delta C_{val}}{\delta \mathbf{x}_c} = -\frac{\delta^2 C_{tr}}{\delta \mathbf{x}_c \delta \mathbf{w}} \cdot \mathbf{z} = -\frac{\frac{\delta C_{tr}}{\delta \mathbf{x}_c}(\mathbf{w} + r\mathbf{z}) - \frac{\delta C_{tr}}{\delta \mathbf{x}_c}(\mathbf{w})}{r} \tag{5.7}$$

taking r is a very small scalar quantity.

5.2 Experiments and Evaluation

5.2.1 Overall Algorithm

This sub-section summarises the overall conjugate-gradient-optimised algorithm used to solve the attacker's optimisation problem, i.e. to generate the optimal attacking point to poison a classifier.

Algorithm 2 Finding the Set of Poisoning Points (Greedy)

```
Require: training dataset $D$, validation dataset $\hat{D}$, size of required poisoning points
    set $np$, learning rate $\alpha$, training (updating) iteration $iter$
 1: $P \leftarrow \{\}$
 2: for $j = 1, ..., np$ do
 3:     $x_c \leftarrow chooseInitialisePoint(D)$
 4:     for $i = 1, ..., iter$ do
 5:         $w \leftarrow trainClassifier(\{D \cup x_c\})$
 6:         $g \leftarrow (\delta C_{val})/(\delta x_c)$                ▷ use findGradient
 7:         $x_c \leftarrow x_c + \alpha g$
 8:         $x_c \leftarrow \Pi_x(x_c)$        ▷ where $\Pi_x(\cdot)$ is the projection operator
 9:                                            ▷ projecting $x_c$ onto the feasible domain
10:     end
11: $D \leftarrow \{D \cup x_c\}$
12: $P_j \leftarrow \{P_j \cup x_c\}$
13: end
14: return $P$
```

Algorithm 3 Finding the Gradient for the Poisoning Point: *findGradient*

```
Require: weight $w$, poisoning point $x_c$, training dataset $D$, validation dataset $\hat{D}$
 1: $A \leftarrow (\delta^2 C_{tr})/(\delta w^2)$
 2: $b \leftarrow (\delta C_{val})(\delta w)$
 3: $z \leftarrow conjugateGradient(A, b)$
 4: $g \leftarrow -((\delta^2 C_{tr})/(\delta x_c \delta w))z$
 5: return $g$
```

Algorithm 4 Initialising Poisoning Point

```
Require: training dataset $D$, label of poisoning point $y_c$
 1: $w \leftarrow trainClassifier(D)$
 2: $\bar{x} \leftarrow$ set of $x \in D$ where $y_j \neq y_c$
 3: $x_{c0} \leftarrow max_{\bar{x}}\{L(w, \bar{x}, y_c)\}$        ▷ L is the classifier's loss function
 4: return $x_{c0}$
```

5.2.2 Correctness Evaluation

The *Synthetic Dataset* is small and only has 2 features. These two characteristics allowed me to generate the poisoning points quickly to test and evaluate whether my implementation of the bi-level optimisation problem solver was correct.

To carry out this correctness evaluation, I plotted a colour map like shown in figure 5.1 to visualise the direction that the poisoning point is updated with gradient descent, against the cost of the bi-level optimisation problem represented by the colour. As observed, the poisoning point was updated towards the area with higher cost for the classifier and ended up at the point which has highest cost within the feasible constraints (represented by the rectangle). This shows that the implemented poisoning algorithm is correct. Figure 5.2 is plotted to show that as the cost is maximised in figure 5.1, the classification rate of the classifier is reduced. Notice that such colour maps could only be generated for problems which use 2-featured dataset such as the Synthetic Dataset.

It is clear from the colour map that the poisoning point was updated towards the area with the highest cost within the limited constraints, which proved the correctness of my bi-level optimisation solver.

5.2.3 Classifier Error Experiment

We can now carry out experiments to examine how well different machine learning classifiers perform under the poisoning attack. I used the Spambase dataset for this experiment because it has a reasonable number of features of 54 – large enough for the problem to not be too simple and small enough that the algorithm to not take too long to run.

As mentioned in section 3.4.2, different learning rates and number of iterations are required to train each classifier in order for them to work well, and this is the same for the update of the poisoning points. Determining the suitable learning rates and iterations by looking at the cost graphs as mentioned in section 3.4.2, I arrived at the values shown below for this experiment.

Classifier	Iter	LR	Inner Iter	Inner LR
ADALINE	2000	0.01	2000	0.1
Logistic Regression	1000	0.1	1500	0.5

Table 5.1: Table showing the hyper-parameters used in the experiment which results shown in figure 5.5, where *Iter* and *LR* are the number of iterations and value of learning rate used in gradient descent of the outer optimisation problem i.e. to update the optimal poisoning point, and *Inner Iter* and *Inner LR* are the same values used for the inner optimisation problem i.e. to find the optimal weights of each classifier to best classify a given training dataset

False Positive and False Negative Rates

In this experiment I had only generated the poisoning points with positive class, and the reason for this was discussed in section 4.1.2. One reasonable way to evaluate the performance of this seemingly *targeted* attack is to look at the *false positive* and *false negative* rates i.e. the rate which the negative class samples are mis-classified as positive class (*ham* as *spam*) and the rate which the positive class samples are mis-classified (*spam* as *ham*), respectively. Intuitively, we would expect the false positive rate to increase and the false negative rate to stay constant because of the nature of an attack targeting the positive class. However, this was not the case.

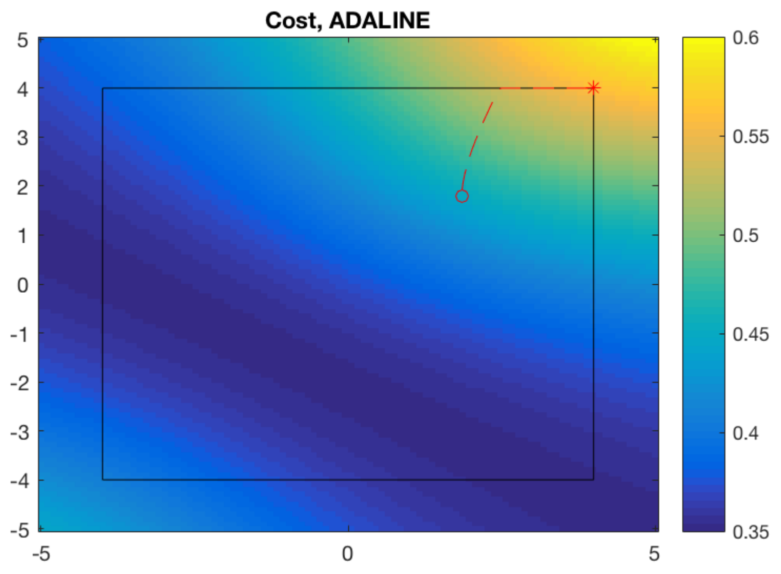


Figure 5.1: Colour map showing the direction of a poisoning as it is updated from the initial point –circle– to the final point –star, where colours represent the cost of training the ADALINE classifier with respect to adding different values poisoning point to the training dataset

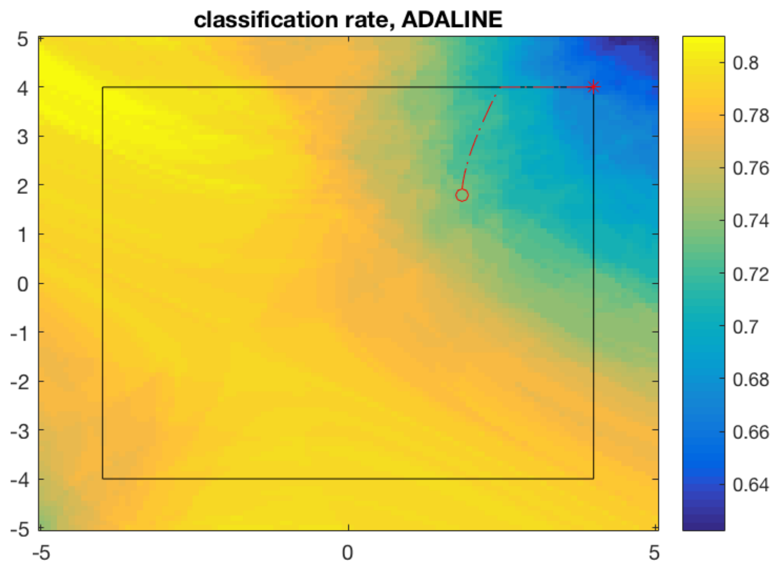


Figure 5.2: Colour map showing the direction of a poisoning as it is updated from the initial point –circle– to the final point –star, where colours represent the classification rate of the ADALINE classifier with respect to adding different values poisoning point to the training dataset

Looking at the false positive and false negative rates of both classifiers (ADALINE and Logistic Regression) under the poisoning attack in figure 5.3 and 5.4, we would see that the false negative rate – although by a smaller amount compared to the false positive rate – had also risen noticeably.

This result suggests that the attack carried out was not a purely a *targeted* attack. By taking a closer look at the bi-level optimisation problem we solved for this experiment, we would be able to discuss the reason behind this behaviour of the increase in the false negative rate.

$$\begin{aligned} \underset{\mathbf{x}_c \in D}{\text{maximise}} \quad & C_{val}(\mathbf{w}, b) \\ \text{subject to} \quad & \mathbf{w}, b = \underset{\mathbf{w},b}{argmin}\, C_{tr}(D, \mathbf{w}, b) \end{aligned} \tag{5.8}$$

With this bi-level optimisation problem, although the attacker only has the capability to inject a poisoning sample $\mathbf{x}_c$ of a positive class, this will not affect the outer optimisation problem which tries to maximise the damage to the classifier indiscriminately i.e. this outer optimisation problem will try to increase the overall mis-classification rate not the mis-classification rate of a certain targeted class. Knowing this, it would be reasonable to evaluate the performance of the attack using the overall classification error.

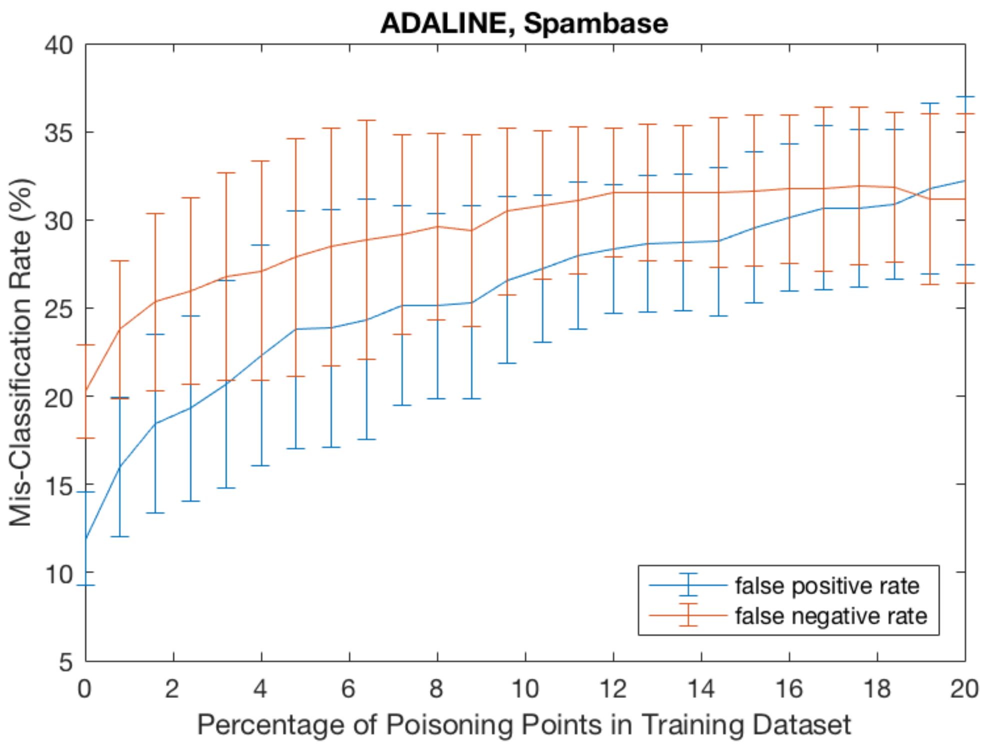


Figure 5.3:

Classification Error

As seen in figure 5.5, I had successfully poisoned the two classifiers. With the proportion of poisoning points of about 4% I had increased the classification error by about 10%. With 20% poisoning points in the training set, ADALINE and Logistic Regression classifiers had 31.8% and 28.2% respectively. Note that with

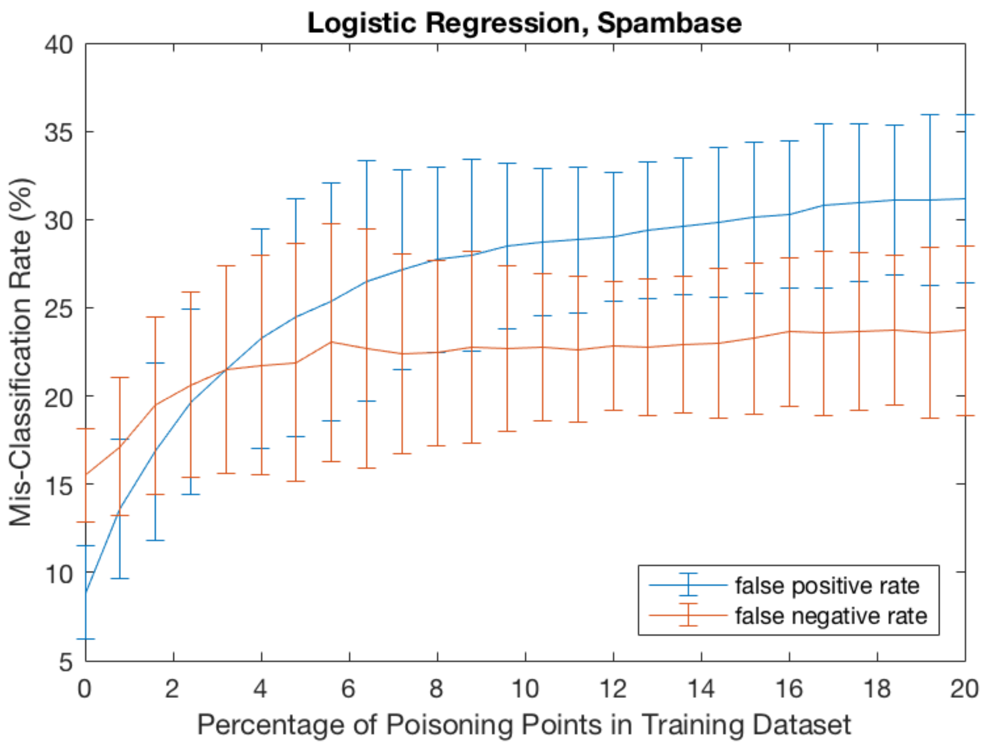


Figure 5.4:

classification error of 50% a classifier is as good as classifying a sample at random. Notice that with no poisoning point, Logistic Regression had performed better with the classification error of 11.48% compared to ADALINE with classification error of 15.19%. This shows that the Logistic Regression is a superior classifier for the Spambase dataset classification problem. However, when the poisoning points are added into the training dataset, both classifiers' classification error increase equally, suggesting that both classifiers suffer equally from the poisoning attack.

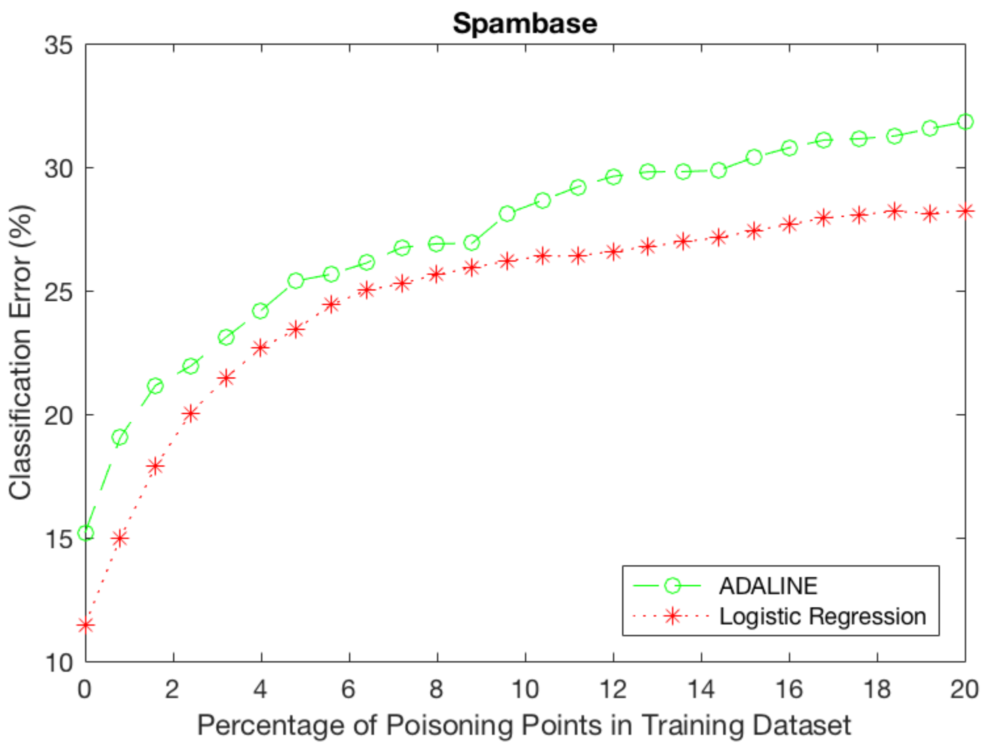


Figure 5.5: Graphs of Error Rate of ADALINE and Logistic Regression Classifiers after being poisoned

Chapter 6

Poisoning with Back-Gradient Method

6.1 Back-Gradient Optimisation

The work of L. Muñoz-González [24], shows that we can solve the attacker's bi-level optimisation problem with another method called the *Back-Gradient Descent*. The main advantage of this method is that, unlike the other methods we have seen so far, it does not require the KKT stationary condition of the inner optimisation problem to be met, i.e. when solving for the optimal weight w, the gradient $\frac{\delta C_{tr}}{\delta \mathbf{w}}$ does not have to be zero. This has enabled us to solve for the bi-level optimisation problem without needing to fully solve the inner-optimisation problem. In our case, we have two direct benefits from this characteristic.

1. **Faster (Better Scalability)**: with back-gradient method, to get the weights $\mathbf{w}$ we want, we would no longer have to train the learning system with the full number of iterations. This would mean solving the bi-level optimisation problem more efficiently.

2. **Enable Attacks on Neural-Network Systems**: in Neural-Network learning systems, the KKT-condition stability condition does not have to be satisfied. The previous methods that have taken advantage of the KKT-condition being satisfied i.e. $\frac{\delta C_{tr}}{\delta \mathbf{w}} = \mathbf{0}$, would not be able to solve for the Neural-Network poisoning points. This is, however, not the case for the back-gradient method as it does not require the KKT-condition to be satisfied. Therefore, back-gradient method allows us to evaluate the effects of the poisoning attack on Neural-Network classifiers.

6.1.1 Algorithm

See algorithm 5.

Algorithm 5 Back-Gradient Descent

```
Require: weight of a system trained after T iterations $w_T$, learning rate $\alpha$, Lost (cost) function of the learning system $L(w, x, y)$, training dataset $D$, validation dataset $\hat{D}$
1: $dx_c \leftarrow 0$ , $dw \leftarrow (\delta C_{val})/(\delta w)$
2: for $t = T, ..., 1$ do
3:     $dx_c \Leftarrow dx_c - \alpha\ dw((\delta^2 C_{tr})/(\delta w\ \delta x_c))$
4:     $dw \Leftarrow dw - \alpha\ dw((\delta^2 C_{tr})/(\delta w\ \delta w))$
5:     $g_{t-1} = (\delta C_{tr})/(\delta w_t)$
6:     $w_{t-1} = w_t + \alpha\ g_{t-1}$
7: end
8: return $dx_c$                    ▷ $dx_c$ is the gradient of $C_{val}$ w.r.t. $x_c$
```

6.2 Experiments and Evaluation

6.2.1 Time Experiment

Test 1 - Effect of Different Classifiers

Figures 6.1 - 6.3 show the time different for each classifier – ADALINE and Logistic Regression – to compute the outer-optimisation problem gradient $\frac{\delta C_{val}}{\delta \mathbf{x}_c}$ for each method – conjugate and back gradient – with each of the 3 datasets.

Observations:

1. Differences of ADALINE and Logistic Regression Classifier do not effect time, i.e. whether we use the ADALINE or Logistic Regression classifiers, the time taken for the gradient computation is similar.

2. As training sample size increases, the time taken for both conjugate-gradient and back-gradient algorithm increases. However, the back-gradient algorithm increases at a much lower rate i.e. having a better scalability.

3. Different datasets causes dramatic time differences. The time taken to compute the gradient with any of the two algorithms range from approximately 0.01-0.17, 0.2-4.3, and 0.2-14 seconds for the Spambase, Ransomeware, and MNIST datasets respectively.

4. The back-gradient algorithm performs better for the two larger-feature datasets *Ransomware* and *MNIST* while performing worse with the *Spambase* dataset which has smaller feature size.

Test 2 - Effect of Training with Different Feature Size

The observation (3) from Test 1 (above) suggests that the number of features in the dataset contribute greatly to how much time each algorithm takes to perform the gradient computation. Knowing this, I have plotted the graph in figure 6.4-6.5 to show the scalability of each algorithm with respect to the number of features in the problem it is trying to solve.

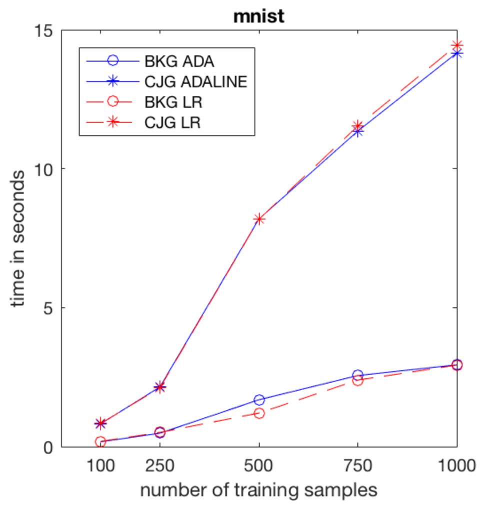


Figure 6.1:

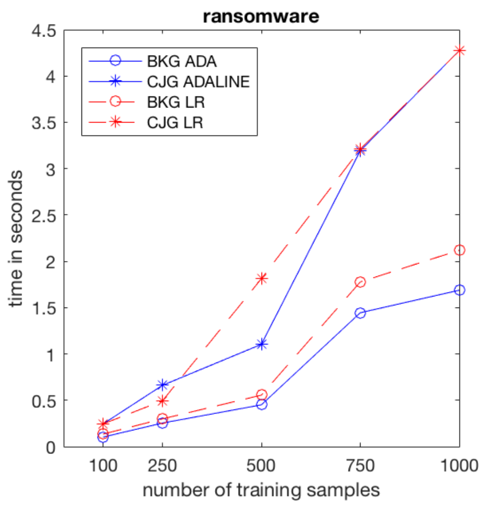


Figure 6.2:

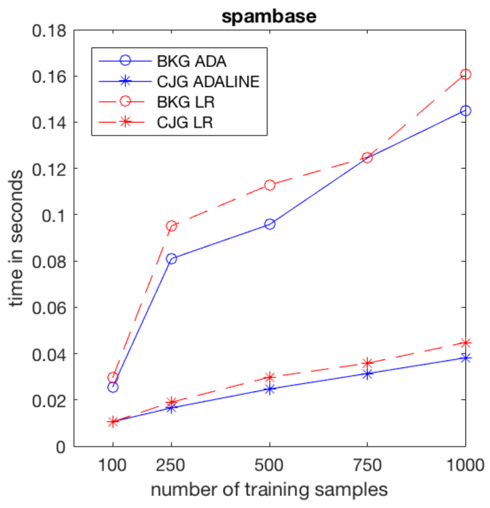


Figure 6.3:

Here we see that the back-gradient algorithm is indeed slightly slower than the conjugate-gradient algorithm when the problem it is solving has small number of features. The graph in figure 6.4-6.5 shows that, as the number of feature grows, the time taken for the back-gradient algorithm to solve for the gradient increases linearly while that of the conjugate-gradient algorithm increases more than linearly. The graph suggests that the back-gradient algorithm performs better when the feature size is higher than approximately 100 features. However, this conclusion cannot be made with the current information as the experiments were carried out with only 3 different size of features; 54, 400, and 784. The lines in between each point are merely results of linear interpolation. The exact behaviour of how each algorithm scales cannot be determined by the shape of the graph by the same reason. Nevertheless, at this point, we can conclude that:

1. the back-gradient algorithm scales better with respect to the number of features in the problem it is solving,

2. the back-gradient algorithm starts performing faster than the conjugate gradient when the feature size of the problem is between 54 and 400 features, where it is very probable that this value is within the range 54-100.

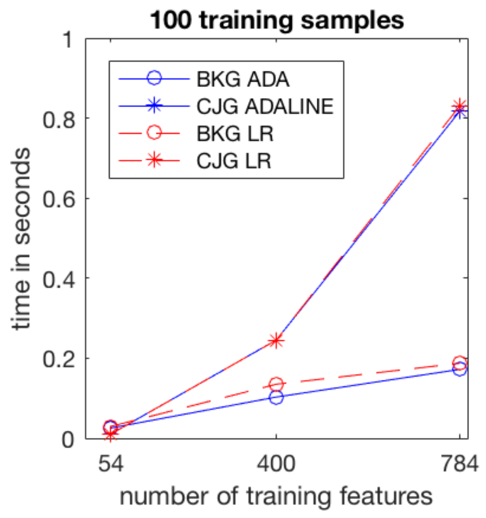


Figure 6.4:

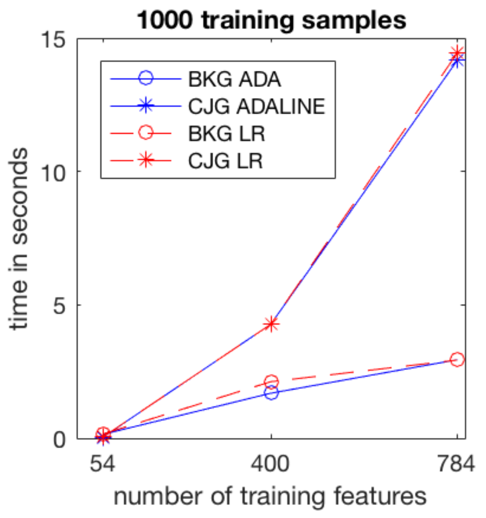


Figure 6.5:

Test 3 - Effect of Different Sizes of Training Data

Figure 6.6-6.7 show the scalability of each method – conjugate-gradient and back-gradient – as the number of training samples and the number of features in the problem increases.

6.2.2 Classification Error

Linear Classifiers

For ADALINE and Logistic Regression, my expectation was that the classification errors would be similar for both back-gradient and conjugate-gradient methods. This was because the two algorithm are eventually solving the same problem mathematically.

Graphs in figure 6.8 and 6.9 shows that the results are as expected. I believe that the reason why the results are not identical despite the two algorithms trying to solve the same equation is because of the effect of the parameters used such as the learning rates and iteration numbers as described in earlier in section 3.4.2.

Neural-Network Classifier

Figure 6.10 shows the multi-layered perceptrons – a neural-network learning system – being poisoned by the back-gradient method. From the graph we can observe a rise in the classification error of about 15% in each problem.

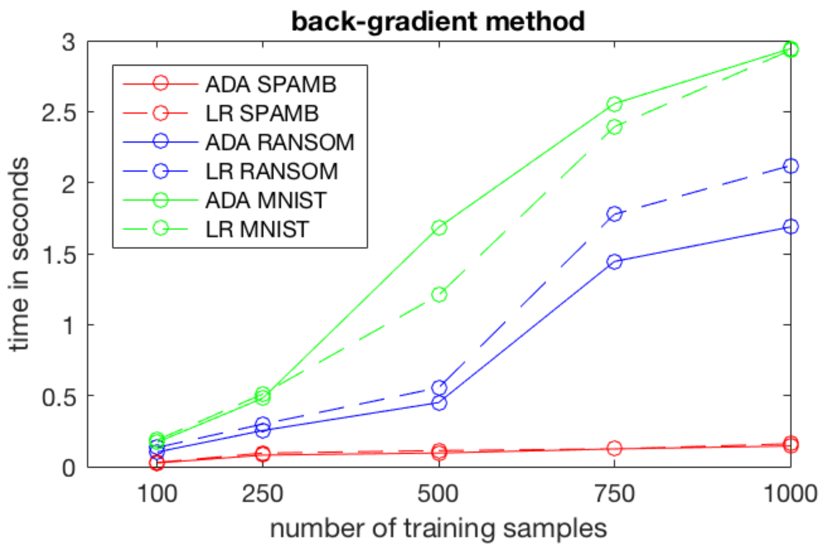


Figure 6.6:

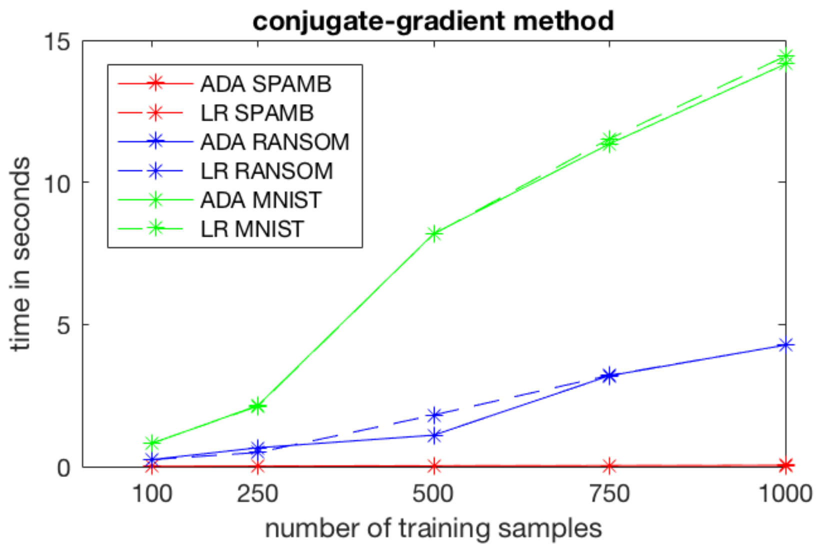


Figure 6.7:

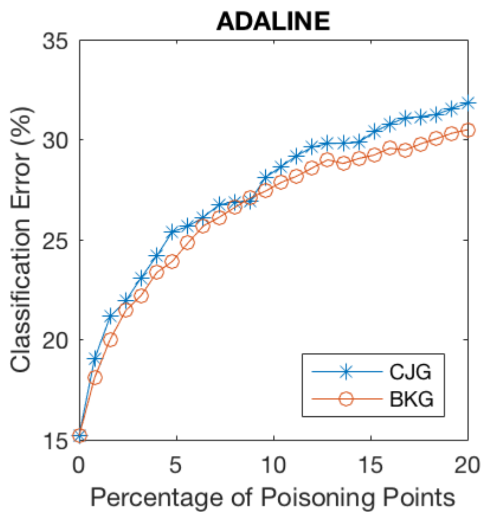


Figure 6.8: with the Spambase Dataset

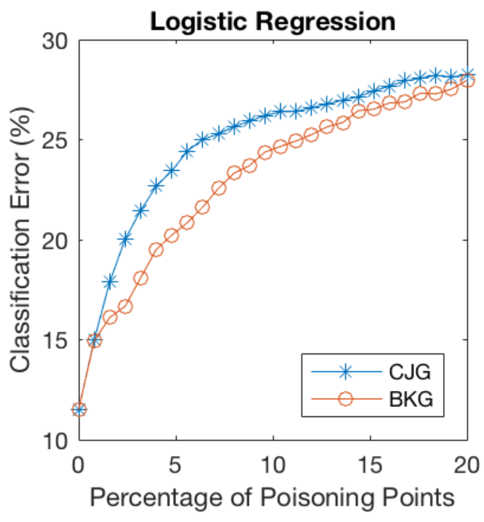


Figure 6.9: with the Spambase Dataset

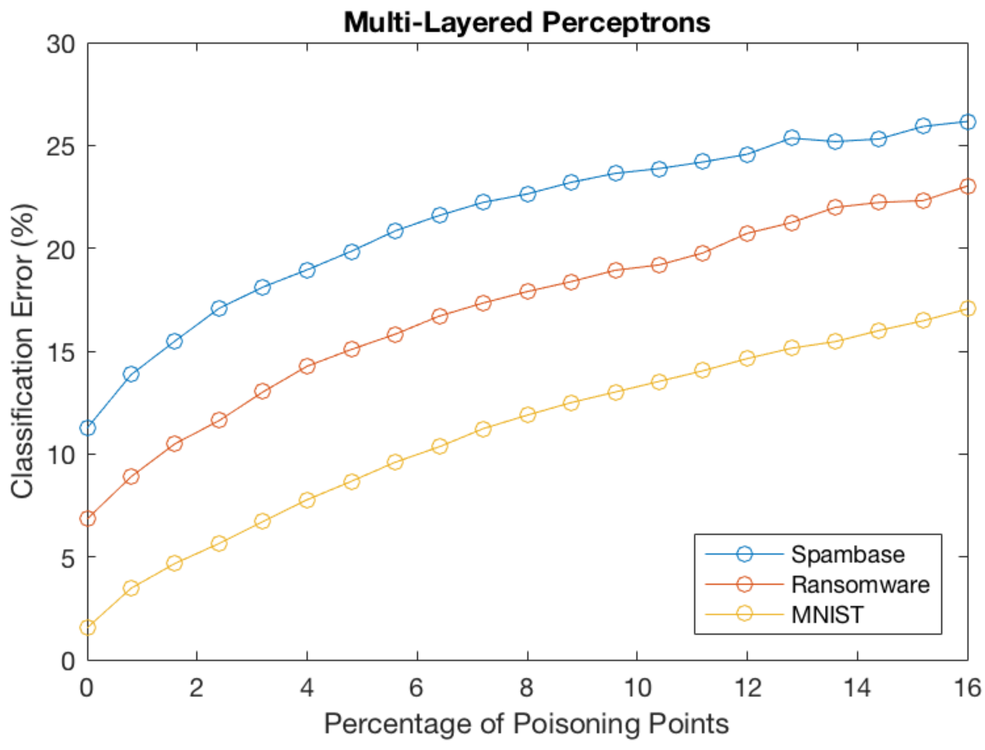


Figure 6.10:

Chapter 7

The Coordinated Attack

In the real-world settings, an attacker would have access to a proportion of the training dataset. This means that an attacker would have to generate a number of poisoning samples to include them into the training dataset. So far, we have only been using a *greedy* method to generate multiple poisoning point – algorithm 2. As shown in algorithm 2, the attacker would solve for one optimal poisoning point, add it into the training dataset, then use the new training set to solve for a new optimal poisoning point.

Although this *greedy* method was shown to have successfully carried out the attack to the three machine learning classifiers – ADALINE, Logistic Regression, and Multi-Layered Perceptron – it was plausible that a non-greedy method would be able to work better i.e. it would be able to generate a higher classification error. This section will discuss about this hypothesis, and will explore the *coordinated attack* strategy – a non-greedy attack method.

7.1 The Algorithm

Algorithm 6 Finding the Set of Poisoning Points (Coordinate Method)

```
Require: training dataset $D$, validation dataset $\hat{D}$, size of required poisoning points
    set $np$, learning rate $\alpha$, training (updating) iteration $iter$
 1: $P \leftarrow chooseMultipleInitialisePoint(D, np)$
 2: for $i = 1, ..., iter$ do
 3:     $w \leftarrow trainClassifier(\{D \cup P\})$
 4:     for $j = 1, ..., np$ do
 5:         $x_c \leftarrow P_j$
 6:         $g \leftarrow (\delta C_{val})/(\delta x_c)$                    ▷ use findGradient
 7:         $x_c \leftarrow x_c + \alpha g$
 8:         $x_c \leftarrow \Pi_x(x_c)$          ▷ where $\Pi_x(\cdot)$ is the projection operator
 9:                                              ▷ projecting $x_c$ onto the feasible domain
10:         $P_j \leftarrow x_c$
11:     end
12: end
13: return $P$
```

Algorithm 7 *chooseMultipleInitialisePoint*

```
Require: training dataset $D$, label of poisoning point $y_c$, number of poisoning
    points $np$
1: $w \leftarrow trainClassifier(D)$
2: $\bar{x} \leftarrow$ set of $x \in D$ where $y_j \neq y_c$
3: $\bar{x}_s \leftarrow$ sort $\{\bar{x}\}$ in descending order of $L(w, \bar{x}, y_c)$
4: return first $np^{th}$ elements of $\bar{x}_s$
```

Notice that instead of computing the gradient $\frac{\delta C_{val}}{\delta \mathbf{x}_c}$ for one point in each iteration as before, this algorithm calculates q gradients each of the q poisoning points in one iteration. One side-effect of this is a faster performance as the same weight $\mathbf{w}$ is used to compute multiple gradients.

7.2 Theory behind the algorithm

The explanation of why this algorithm could produce better results than the greedy algorithm is the idea that many poisoning points could work together to achieve a common goal. This fits better to the problem of *injecting a set of poisoning points* to poison a classifier. Looking at the optimisation problem, the greedy method would be solving the following problem q number of times:

$$\underset{x_c \in D}{\text{maximise}} \quad C_{val}(\mathbf{w}, b) \tag{7.1}$$

while the coordinated method would be solving the following problem once:

$$\underset{\{\mathbf{x}_c\}_{j=1}^{q} \in D}{\text{maximise}} \quad C_{val}(\mathbf{w}, b) \tag{7.2}$$

As seen in the equations, when trying to find the optimal set of poisoning points, the greedy method is not strictly solving for that goal. Instead, it only tries to find a single poisoning point that maximises the cost C_{val}. The coordinated method, on the other hand, tries to look for the q poisoning points that maximises C_{val}, conforming to the goal of the attacker.

From this, it is clear that the two methods would produce a different sets of poisoning points. Thus, it is worth comparing the effects of each method in poisoning the classifiers.

7.3 Experiments and Evaluation

This section shows the experiment carried out to show the classification errors of different classifiers against a different numbers of poisoning points generated by the two algorithms – greedy and coordinate.

7.3.1 Results - Spambase

The result for this experiment in the spam email classification problem is shown in figure 7.1-7.3.

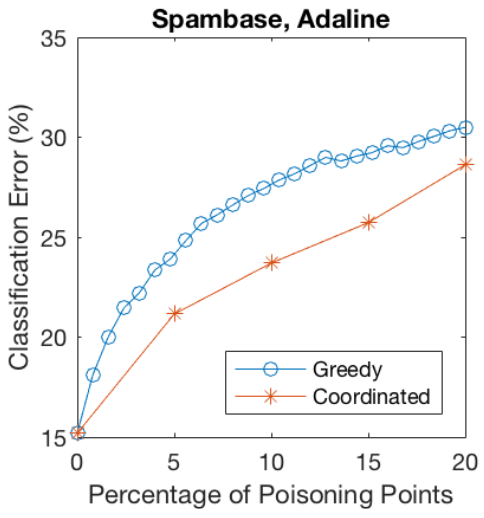


Figure 7.1:

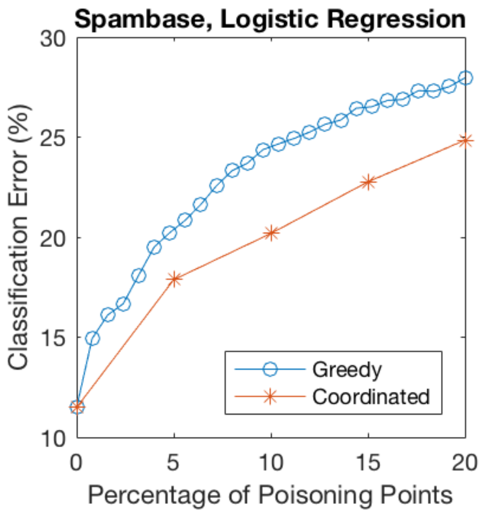


Figure 7.2:

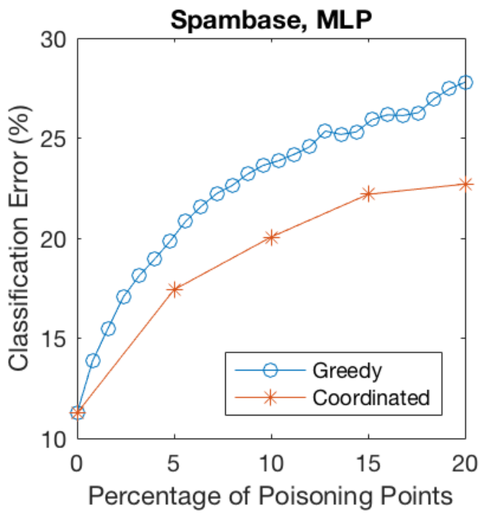


Figure 7.3:

7.3.2 Results - Ransomware

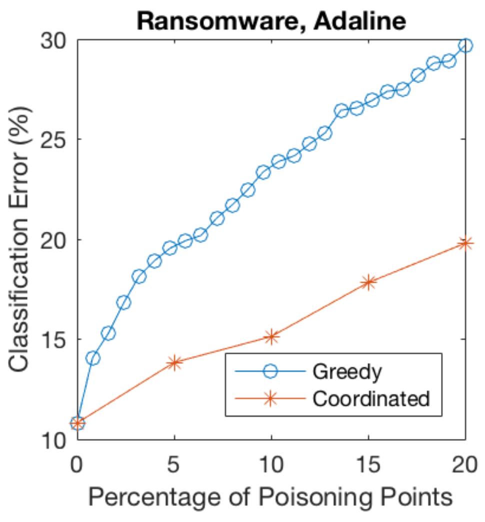
The result for this experiment in the ransomware classification problem is shown in figure 7.4-7.6.

7.3.3 Results - MNIST

The result for this experiment in the hand-written digits classification problem is shown in figure 7.7-7.9.

7.3.4 Result Discussion

As shown, the results we observed did not agree with the hypothesis we made. There is no clear sign that the coordinate method outperform the greedy method. Unlike the other results, results for the MNIST dataset in figures 7.7 and 7.8 show



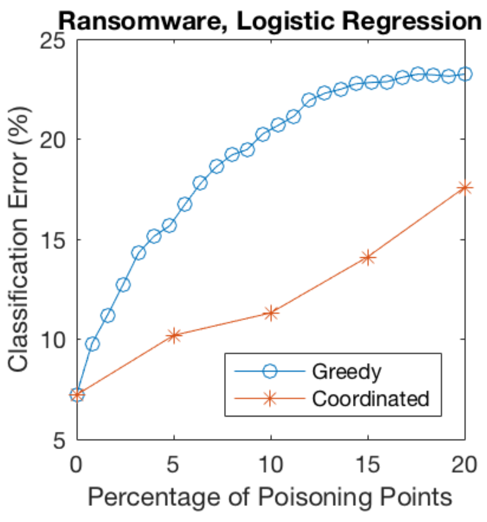


Figure 7.4:

Figure 7.5:

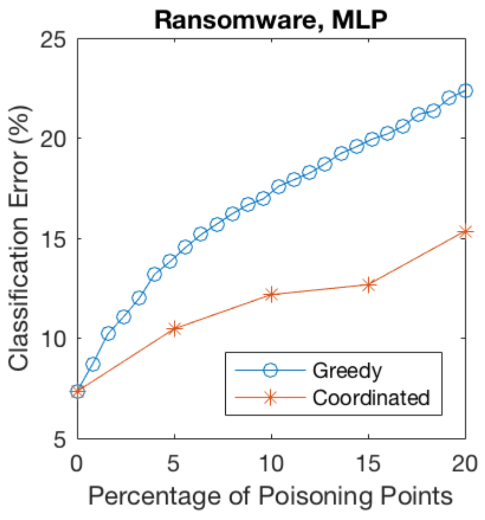


Figure 7.6:

that the performance of the coordinate method is on par with the greedy method. This suggests that the parameters for the coordinate method might have been chosen poorly for the other experiments causing it to perform worse than the greedy method in those experiments. As discussed in section 3.4.2, choosing the optimal parameters is a difficult problem.

It is also possible that the algorithm 6 used in the experiment does not estimate the solution for the optimisation problem described in equation 7.2 well. One example of a minor flaw in the algorithm is that at each iteration, the algorithm solves for $\frac{\delta C_{val}}{\delta \mathbf{x}_c^i}$ using the weight $\mathbf{w}$ found by training the machine learning classifier with $\{D \cup \{\mathbf{x}_c\}_{i=1}^q\}$ of the previous iteration. This could be improved by re-training the classifier with the new $\mathbf{x}_c$ found within one iteration. Thereby, modifying the algorithm to be the algorithm 8.

The extent of how much this moderation in the algorithm would affect the performance could be explored in the future. For the time being, we cannot conclude

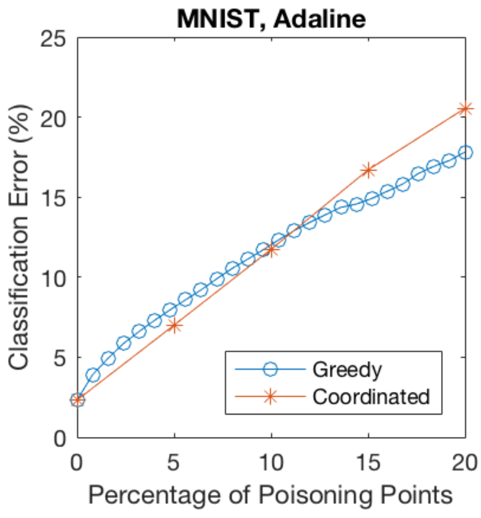


Figure 7.7:

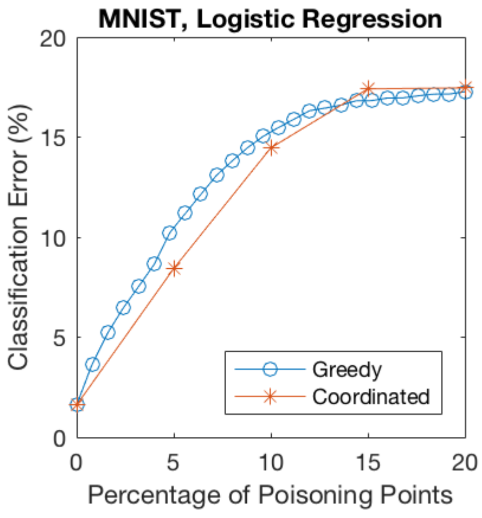


Figure 7.8:

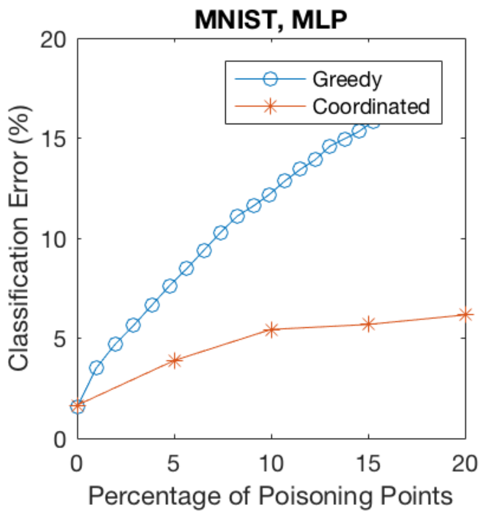


Figure 7.9:

that the coordinate method is superior to the greedy one.

Algorithm 8 Possible Improved Coordinated Method

```
Require: training dataset $D$, validation dataset $\hat{D}$, size of required poisoning points
    set $np$, learning rate $\alpha$, training (updating) iteration $iter$
 1: $P \leftarrow chooseMultipleInitialisePoint(D, np)$
 2: for $i = 1, ..., iter$ do
 3:     for $j = 1, ..., np$ do
 4:         $w \leftarrow trainClassifier(\{D \cup P\})$
 5:         $x_c \leftarrow P_j$
 6:         $g \leftarrow (\delta C_{val})/(\delta x_c)$                    ▷ use findGradient
 7:         $x_c \leftarrow x_c + \alpha g$
 8:         $x_c \leftarrow \Pi_x(x_c)$          ▷ where $\Pi_x(\cdot)$ is the projection operator
 9:                                   ▷ projecting $x_c$ onto the feasible domain
10:         $P_j \leftarrow x_c$
11:     end
12: end
13: return $P$
```

Chapter 8

Poisoning Multi-Class Classifier

8.1 Multi-class Classifier Poisoning

Examining the effect of the poisoning attack on a multi-class classification problem has not been seen before in the literature. However, there is no reason to believe that the same methods we have seen before such as the conjugate-gradient or back-gradient methods cannot be applied to poison a multi-class classifier. It is therefore interesting to examine the performance of our poisoning attack on such classifier, making a contribution with respect to the state-of-the-art.

In this work, we will investigate the effect of the greedy-back-gradient poisoning attack against a multi-class logistic regression classifier. Since the classifier now have to deal with multiple labels for each dataset sample, the *cost function* and the *prediction function* of the binary-class logistic regression classifier has to be modified [13].

Cost Function

The binary-class cost function of the logistic regression classifier will not work for the multi-class case. Instead, the multi-class *Cross-Entropy Function* is used, as it support multiple labels i.e. label vector $\mathbf{y}$.

$$Cost(\mathbf{w}) = \sum_{i=1}^{n} \sum_{k=1}^{C} \mathbf{y}_k \log(Pr(G(\mathbf{x}_i) = k | \mathbf{x} = \mathbf{x}_i)) \tag{8.1}$$

and

$$Pr(G(\mathbf{x}) = k | \mathbf{x} = \mathbf{x}_i) = \frac{e^{\mathbf{w}_k^T \mathbf{x}_i}}{\sum_{k=1}^{C} e^{\mathbf{w}_k^T \mathbf{x}_i}} \tag{8.2}$$

where C is the total number of classes, and $G(\mathbf{x}_i)$ is the *prediction*, and thus $Pr(G(\mathbf{x}) = k | \mathbf{x} = \mathbf{x}_i)$ would translate to "*the probability that the classifier will classify the sample* $\boldsymbol{x}$ *as class k*". This is also known as the *softmax* function.

Prediction Function

In the binary-class case, the logistic regression would have a scalar output value $o = \mathbf{w}^T\mathbf{x}$ and the prediction function would be the sigmoid function which would, in practice, result in 0 or 1. However, in the multi-class case, the output values would be a vector $\mathbf{o}$ where each dimension corresponds to each class. For class k,

$o_k = Pr(G(\mathbf{x}) = k|\mathbf{x} = \mathbf{x}_i)$. With output being a vector, our classifier would predict the class of $\mathbf{x}$ by choosing the value k that corresponds to the highest value in the output vector $\mathbf{o}$.

$$G(\mathbf{x}) = \underset{k}{\text{argmax}}\ o_k \tag{8.3}$$

Finding Optimal Weights

The weights of the multi-class logistic regression system is optimised using the standard gradient descent technique, similar to the case of the binary-class logistic regression.

8.2 Experiments and Evaluation

This section will show the results of the experiments that I have run to evaluate the effects of the poisoning attack on a multi-class classifier. The multi-class Logistic Regression was used for all experiments in this section. It was tested against, first the Matlab IRIS dataset, then the multi-class MNIST dataset (1000 samples training set, 1000 samples in validation set, and 8000 samples in the test set). Since the number of the training set increased to 1000 – from 100 in all previous experiments – in each of our 10 dataset split described in 9.2, and due to the constraints in time and computational power, I decided to stop the experiment after generating 60 poisoning points in each split, creating the set of poisoning points up to about 6%. The IRIS dataset was used to test if the code was running correctly as it is smaller in the number of features and training size, and thus it is much faster to train with. In this section we will not show the result of the IRIS dataset, but will go straight to examining the multi-class MNIST problem. In the following experiments, since we cannot conclude that the superiority of the *coordinate* method, I have continued using the *greedy* method to generate the poisoning points.

Targeted Attack VS Indiscriminate Attack

The experiments carried out will examine two different type of attacks: targeted and indiscriminate. The main difference in these two attack strategies are:

- Targeted attack chooses the initial value of the poisoning point by randomly selecting the value of a training sample with a given label, while the indiscriminate attack select a random value of a training sample in the whole training set. In our experiments, I chose the initial label as 3.
- Targeted attack generates poisoning points and label it with the same targeted label, while the indiscriminate attack calculates for the label that generates highest cost with respect to the chosen initial poisoning point value. In our experiments, I chose the targeted label as 8.

See Algorithms 9 and 10.

Algorithm 9 Initialising Multi-class Poisoning Point (Targeted)

```
Require: training dataset $D$, initial label $y_i$, target label $y_t$
1: $w \leftarrow trainClassifier(D)$
2: $\bar{x} \leftarrow$ set of $x \in D$ where its corresponding $y = y_i$
3: $x_{c0} \leftarrow selectRandom(\bar{x})$
4: $y_{c0} \leftarrow y_t$
5: return $x_{c0}$, $y_{c0}$
```

Algorithm 10 Initialising Multi-class Poisoning Point (Indiscriminate)

```
Require: training dataset $D$, set of all class labels $C$, cost function $L(w, x, y)$
1: $x_{c0} \leftarrow selectRandom(D)$
2: $y_{c0} \leftarrow argMax(L(w, x, y), C)$          ▷ argMax outputs label $y \in C$ that
3:                                                    ▷ gives highest cost $L(w, x_{c0}, y)$
4: return $x_{c0}$, $y_{c0}$
```

8.2.1 Classification Error

To evaluate the performance of the attack on the classifier, we will begin by looking at the classification error rate of the classifier as the number of poisoning points grows.

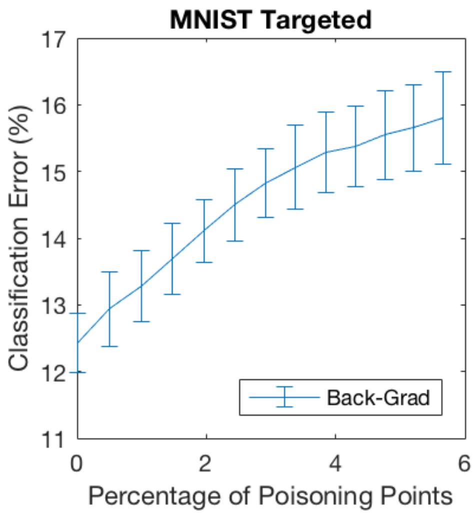


Figure 8.1:

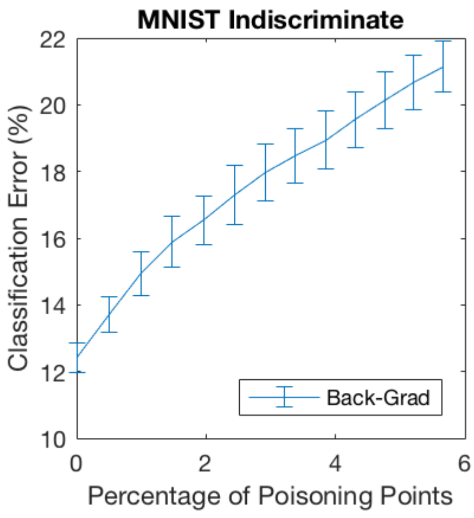


Figure 8.2:

As shown in figures 8.1 and 8.2, poisoning of the multi-class Logistic Regression classifier for MNIST dataset is promising in both targeted and indiscriminate case. The graphs shown constant increase in classification error rate as the number of the poisoning points increases. We can observe approximately 3% and 9% increase in the targeted and indiscriminate case respectively after 6% of training data have been poisoned.

To further examine the effect of each attack on the classification problem, we can compute the classification rate of each class from the confusion matrices. Figures 8.3 and 8.4 show classification rates of each class while being poisoned. As we might have expected, the indiscriminate attack raises the classification error of all classes, while the targeted attack only raise the classification error of certain classes i.e. 8 and 3. However, it is interesting to observe that for both attack strategies, class 5 is more prone to error whilst class 1 is less prone to error to the other classes. Increases in classification rates of class other than the two targeted classes (3 and 8) suggest the similarities in the features of those classes to that of class 3 and 8.

8.2.2 Confusion Matrices

For multi-class classifier problems, it is worth looking at the confusion matrix to examine further how the classifier classify samples from each class. Figures 8.5 -8.7 and 8.11 - 8.14 shows the extended version of the confusion matrices we got from this experiment. In this matrix we have extended the original confusion matrix by dividing each cell by the total number of samples belonging to its actual class then times 100. Values in this matrix will show the rate(%) which a sample of an actual class is classified as a predicted class, representing classification rates for the diagonal cells and mis-classification rates for the non-diagonal cells.

For the indiscriminate case, figure 8.13 shows no interesting feature except that all the diagonal cells have been decreased i.e. classification rate of all classes are reduced. The increase in classification error – diagonal cells – seems to be at random. On the other hand, looking the similar matrix for the targeted case in figure 8.9, we can clearly see that the increase in classification errors relating to class 8 – yellow stripes across row 8 and column 8. This is the result of all other classes being mis-classified as class 8, and the actual class 8 being mis-classified as other classes. We can also observe the fall of classification rates in 3 classes represented by the blue diagonal cells for class 3, 8 and 5 in descending order.

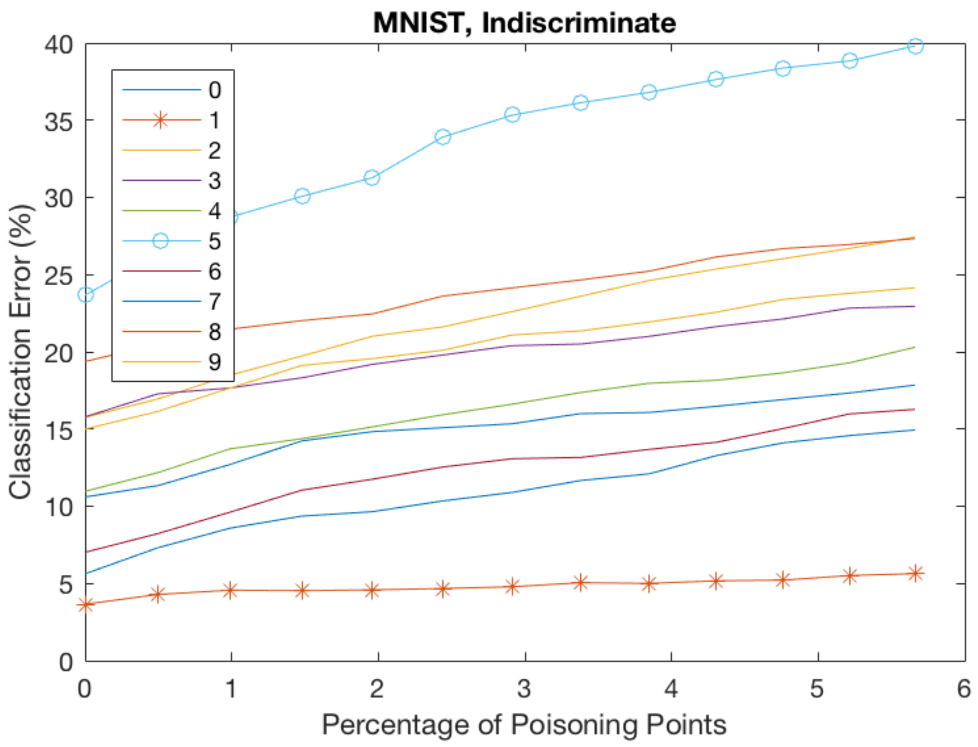


Figure 8.3:

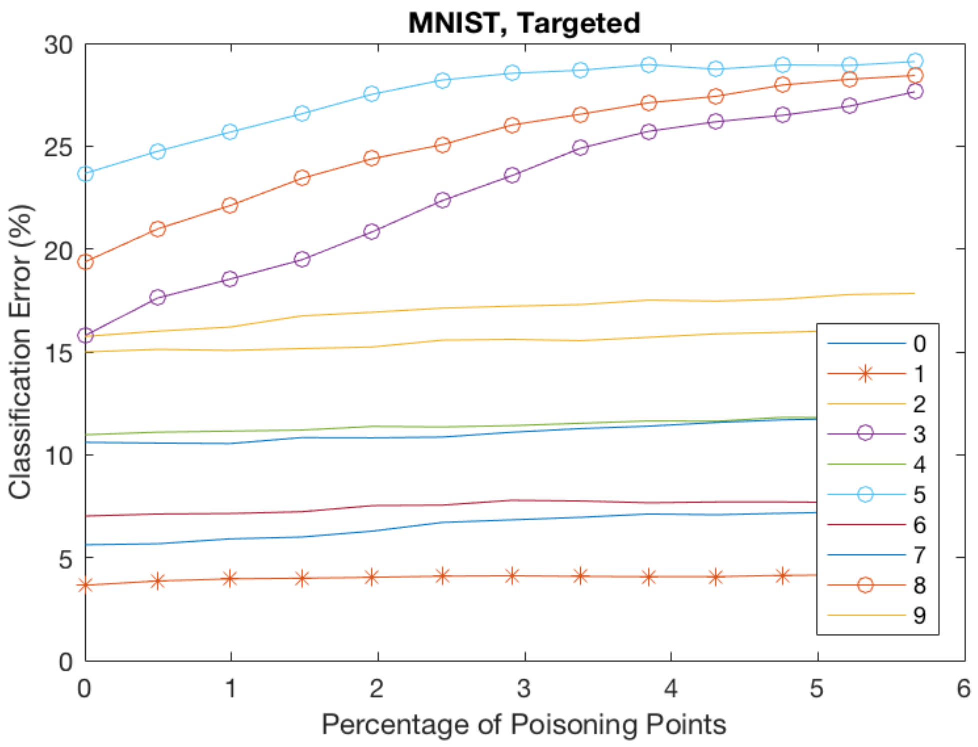


Figure 8.4:

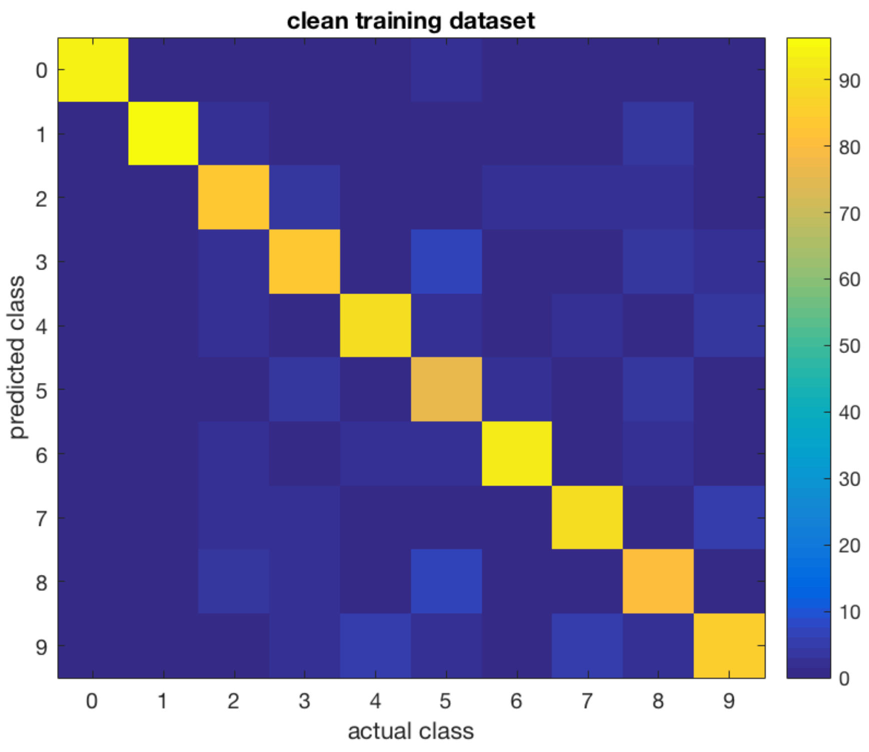


Figure 8.5: A colourmap representation of the confusion matrix of the multi-class Logistic Regression classifier trained with a clean dataset. The colours represent the classification and mis-classification rate (%) i.e. confusion matrix cell value divided by total number of sample from the corresponding actual class.

```
 94.3759         0    1.1942    0.3762    0.0764    1.6411    0.7033    0.5956    1.0267    1.0588
       0   96.3429    1.8167    1.2845    0.9288    0.8398    0.5700    1.3026    3.0247    0.6241
  0.7736    0.5985   84.2488    3.0799    1.1955    1.1466    1.5311    1.6198    2.6110    0.7508
  0.7742    0.3209    1.8228   84.2115    0.2546    6.5562    0.2109    0.2688    3.1432    1.8394
  0.2128    0.1215    1.9669    0.1817   89.0305    2.2378    0.9877    1.6931    0.9206    4.0211
  1.2129    0.7641    0.6013    4.4706    0.1765   76.3302    1.5761    0.1649    4.1266    0.6609
  1.0875    0.1551    2.2962    0.6527    1.6412    2.0349   92.9802    0.0117    1.6759    0.0643
  0.3994    0.4105    2.0450    1.6593    0.7216    0.4483    0.1226   89.3983    0.7382    5.1005
  1.0009    1.1425    3.2346    1.9661    0.6245    6.6403    1.2440    0.2929   80.6296    0.8759
  0.1627    0.1440    0.7735    2.1176    5.3502    2.1249    0.0741    4.6524    2.1037   85.0044
```

Figure 8.6: Confusion matrix of multi-class Logistic Regression classifier when trained with a clean dataset. Each value is the classification or mis-classification rate (%) i.e. confusion matrix cell value divided by total number of sample from the corresponding actual class, and the row and column represents the predicted class and actual class respectively.

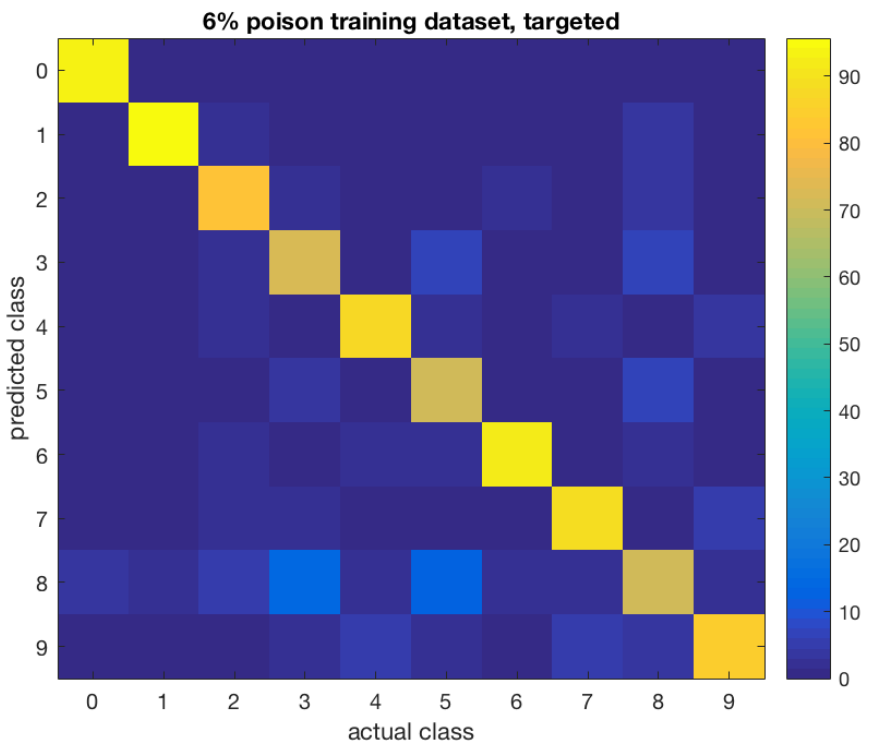


Figure 8.7: A colourmap representing the confusion matrix of multi-class Logistic Regression classifier trained with a 6% poisoned dataset targeting class 3, initialising poisoning points from class 8. The colours represent the classification and mis-classification rate (%) i.e. confusion matrix cell value divided by total number of sample from the corresponding actual class.

```
92.7190     0.0112     1.2212     0.2911     0.1138     1.3480     0.7639     0.6313     0.9472     1.0852
      0    95.6737     1.6989     1.1633     0.9682     0.8994     0.5076     1.1989     3.4989     0.6760
 0.7245     0.6999    82.1585     2.6229     1.1321     1.0463     1.6427     1.3191     3.5798     0.8147
 0.4361     0.4328     1.5029    72.3610     0.2275     6.6863     0.0876     0.3504     6.4654     1.4933
 0.2003     0.1214     1.8875     0.1455    88.1174     2.1919     0.9260     1.7164     1.0518     4.1074
 1.1386     0.8525     0.4308     4.3821     0.2663    70.8900     1.4632     0.1761     6.6216     0.7491
 0.9626     0.0992     2.4617     0.7734     1.5259     2.1661    92.2957          0     2.0453     0.0770
 0.3368     0.4208     2.0099     1.6357     0.7236     0.5801     0.0981    88.2072     1.1459     4.8102
 3.4072     1.5334     5.6783    14.5326     1.5742    12.2305     2.1660     1.9104    71.5679     2.2032
 0.0750     0.1551     0.9503     2.0925     5.3509     1.9614     0.0493     4.4903     3.0762    83.9841
```

Figure 8.8: Values of the confusion matrix shown in figure 8.7. Each value is the classification or mis-classification rate (%) i.e. confusion matrix cell value divided by total number of sample from the corresponding actual class, and the row and column represents the predicted class and actual class respectively.

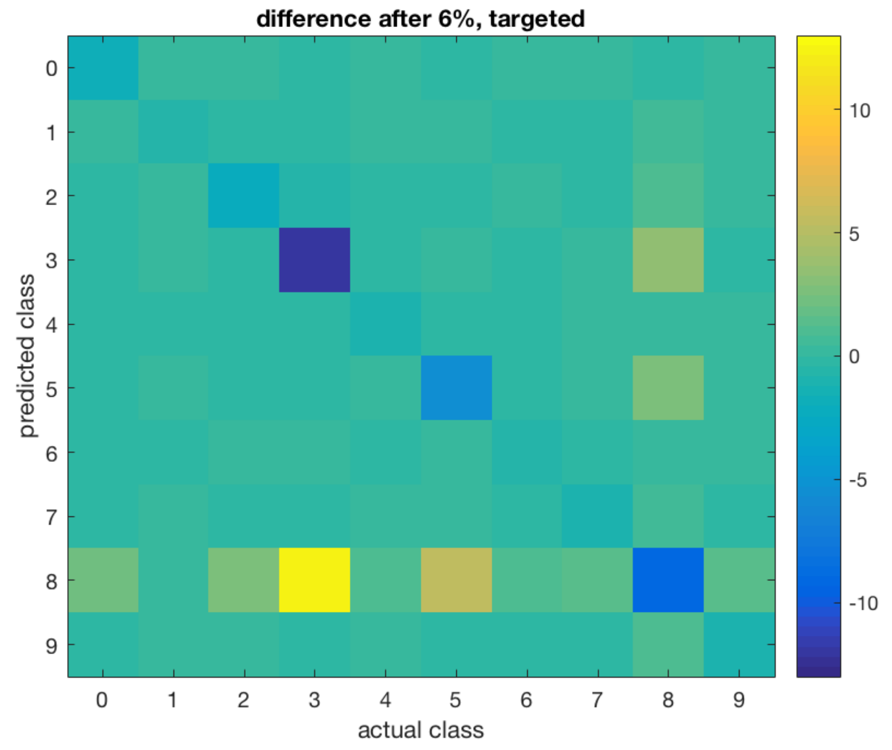


Figure 8.9: This matrix is the difference in the classification and mis-classification rates after the classifier has been poisoned with the targeted attack i.e. the difference between the figure 8.7 and 8.5

```
 -1.6570    0.0112    0.0270   -0.0852    0.0374   -0.2931    0.0606    0.0358   -0.0794    0.0264
       0   -0.6692   -0.1178   -0.1211    0.0394    0.0597   -0.0624   -0.1037    0.4742    0.0519
 -0.0491    0.1015   -2.0903   -0.4570   -0.0634   -0.1003    0.1116   -0.3007    0.9688    0.0639
 -0.3381    0.1119   -0.3199  -11.8505   -0.0271    0.1301   -0.1233    0.0816    3.3222   -0.3461
 -0.0125   -0.0002   -0.0794   -0.0362   -0.9131   -0.0459   -0.0617    0.0233    0.1312    0.0864
 -0.0743    0.0884   -0.1704   -0.0885    0.0898   -5.4402   -0.1129    0.0112    2.4950    0.0881
 -0.1250   -0.0559    0.1655    0.1206   -0.1153    0.1313   -0.6845   -0.0117    0.3694    0.0127
 -0.0626    0.0103   -0.0351   -0.0235    0.0020    0.1318   -0.0245   -1.1911    0.4077   -0.2903
  2.4063    0.3909    2.4437   12.5666    0.9497    5.5902    0.9220    1.6175   -9.0617    1.3273
 -0.0877    0.0111    0.1768   -0.0251    0.0006   -0.1635   -0.0248   -0.1621    0.9725   -1.0203
```

Figure 8.10: Values of the confusion matrix shown in figure 8.9. Each value is the classification or mis-classification rate (%) i.e. confusion matrix cell value divided by total number of sample from the corresponding actual class, and the row and column represents the predicted class and actual class respectively.

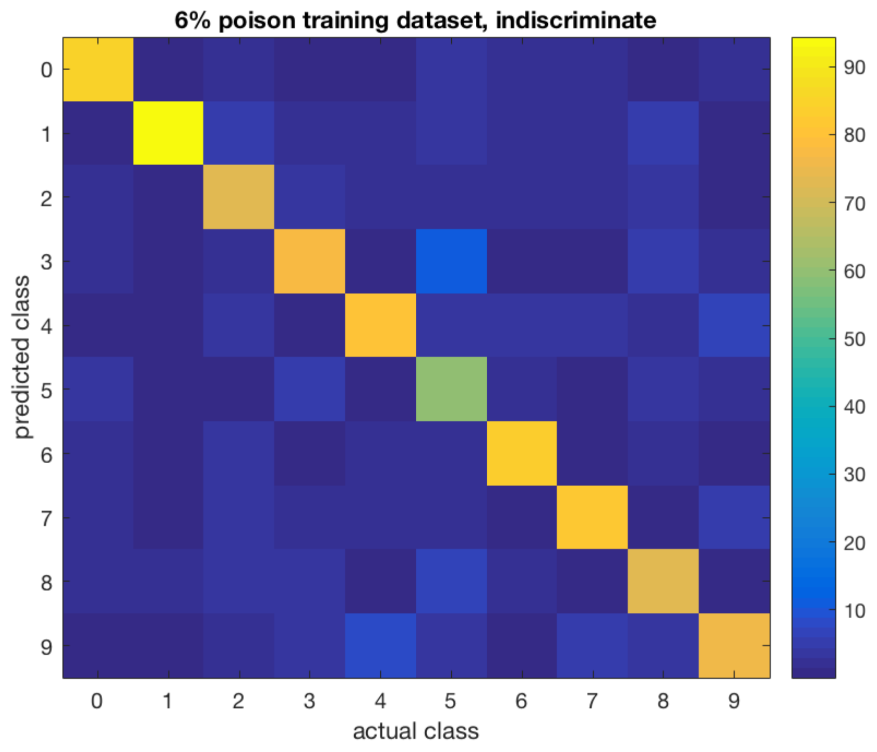
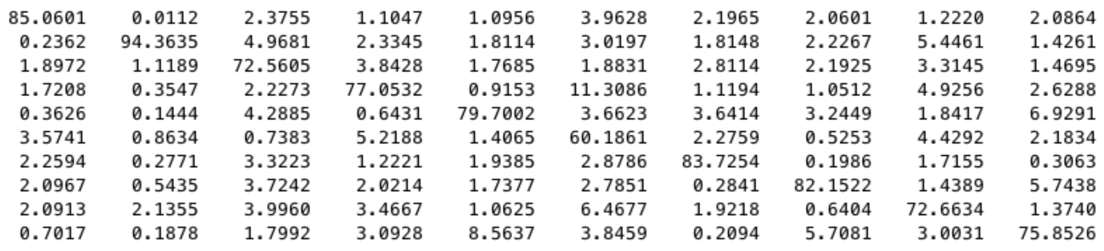


Figure 8.11: A colourmap representing the confusion matrix of multi-class Logistic Regression classifier trained with a 6% poisoned dataset where the poisoning points are made for the indiscriminate attack. The colours represent the classification and mis-classification rate (%) i.e. confusion matrix cell value divided by total number of sample from the corresponding actual class.



```
85.0601    0.0112    2.3755    1.1047    1.0956    3.9628    2.1965    2.0601    1.2220    2.0864
 0.2362   94.3635    4.9681    2.3345    1.8114    3.0197    1.8148    2.2267    5.4461    1.4261
 1.8972    1.1189   72.5605    3.8428    1.7685    1.8831    2.8114    2.1925    3.3145    1.4695
 1.7208    0.3547    2.2273   77.0532    0.9153   11.3086    1.1194    1.0512    4.9256    2.6288
 0.3626    0.1444    4.2885    0.6431   79.7002    3.6623    3.6414    3.2449    1.8417    6.9291
 3.5741    0.8634    0.7383    5.2188    1.4065   60.1861    2.2759    0.5253    4.4292    2.1834
 2.2594    0.2771    3.3223    1.2221    1.9385    2.8786   83.7254    0.1986    1.7155    0.3063
 2.0967    0.5435    3.7242    2.0214    1.7377    2.7851    0.2841   82.1522    1.4389    5.7438
 2.0913    2.1355    3.9960    3.4667    1.0625    6.4677    1.9218    0.6404   72.6634    1.3740
 0.7017    0.1878    1.7992    3.0928    8.5637    3.8459    0.2094    5.7081    3.0031   75.8526
```

Figure 8.12: Values of the confusion matrix shown in figure 8.11. Each value is the classification or mis-classification rate (%) i.e. confusion matrix cell value divided by total number of sample from the corresponding actual class, and the row and column represents the predicted class and actual class respectively.

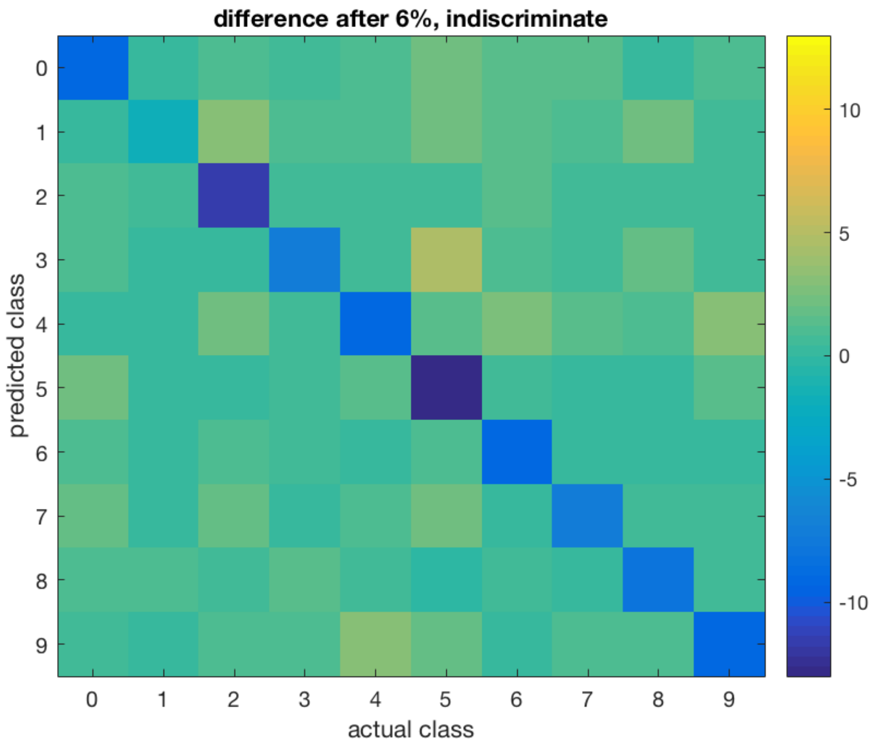


Figure 8.13: This matrix is the difference in the classification and mis-classification rates after the classifier has been poisoned with the indiscriminate attack i.e. the difference between the figure 8.11 and 8.5

```
 -9.3158    0.0112    1.1814    0.7285    1.0191    2.3216    1.4932    1.4646    0.1953    1.0276
  0.2362   -1.9794    3.1515    1.0500    0.8826    2.1799    1.2448    0.9241    2.4215    0.8020
  1.1235    0.5204  -11.6883    0.7629    0.5730    0.7365    1.2803    0.5727    0.7034    0.7187
  0.9466    0.0338    0.4045   -7.1582    0.6607    4.7524    0.9085    0.7824    1.7824    0.7894
  0.1498    0.0228    2.3216    0.4613   -9.3303    1.4246    2.6537    1.5518    0.9211    2.9081
  2.3613    0.0993    0.1371    0.7482    1.2300  -16.1440    0.6998    0.3604    0.3027    1.5225
  1.1718    0.1220    1.0261    0.5693    0.2973    0.8438   -9.2548    0.1870    0.0396    0.2420
  1.6973    0.1330    1.6792    0.3621    1.0162    2.3368    0.1615   -7.2461    0.7007    0.6434
  1.0904    0.9931    0.7613    1.5007    0.4380   -0.1726    0.6778    0.3475   -7.9661    0.4981
  0.5389    0.0438    1.0257    0.9752    3.2135    1.7210    0.1353    1.0557    0.8994   -9.1518
```

Figure 8.14: Values of the confusion matrix shown in figure 8.13. Each value is the classification or mis-classification rate (%) i.e. confusion matrix cell value divided by total number of sample from the corresponding actual class, and the row and column represents the predicted class and actual class respectively.

Chapter 9

Transferability

9.1 Transferability of Attacks

So far I have only tried to poison my local classifier systems. Knowing exactly the code of those classifiers enabled me to generate the poisoning points that are specifically targeting those classifier systems. Although this is a good way to evaluate test robustness of each classifier algorithm, it is unlikely that a real-world attacker would know the exact way that his targeted classifier is coded. For example, he might not know the exact learning algorithm of the system, or how the learning process of the system is optimised.

In the real-world, the attacker would have some knowledge about the system he is targeting. Thus he could model his own version of the targeted classifier system – similar to what I have done with ADALINE and Logistic Regression – and use the model system to generate the poisoning points. It is, therefore, interesting to observe how the poisoning points generated by our methods perform against classifiers made by somebody else, i.e. investigating the transferability of the attacks.

In this experiment I have 9 sets of the poisoning points generated with the greedy algorithm, 3 binary-class datasets (MNIST, Ransomware, and Spambase), and 3 model classifier (ADALINE, Logistic Regression, and Multi-Layered Perceptrons). For example, the *'Spambase ADALINE poison'* is a poisoning point generated to poison our ADALINE classifier for the Spambase problem. With these poisoning points, I have targeted two following classifiers implemented by MathWorks:

1. **The `glmfit` function** - A logistic regression classifier from the Matlab library. This experiment will show that even though the target system matches our model system (logistic regression), differences their implementation will cause the result of the poisoning attack to be different.

2. **The `svmtrain` function** - An SVM classifier from the Matlab library. With this experiment, we will observe the effect of poisoning points generated based on a model system that does not match the target system.

9.2 Experiments and Evaluation

In order to carry out the experiments described in the previous section (9.1) in a way such that the results can be compared with all our previous results in a fair way, I have used the same 10-Splits method as described in section 3.2.2, and the same real 10-Splits datasets as used in the experiments in all previous chapters. Each split of the training set is injected with poisoning points generate for that particular training set split, in other words, the 10 different dataset splits are injected with different poisoning points.

9.2.1 Poisoning Matlab's Logistic Regression Classifier

Figures 9.1-9.3 shows the result of using the different sets of poisoning points to poison the Logistic Regression classifier from Matlab library. From the graph, it is clear that the poisoning points generated by all 3 model systems can be used to poison `glmfit` classifier system, as seen in the increase in classification error rates. It is interesting to see that we can increase the classification error for the MNIST problem by as much as 30%. However, for both MNIST and Ransomware problem, the graphs are not smooth. This suggests that the classifier is not stable for the two datasets. The further investigation on this requires studying the `glmfit` implementation which could be done in the future.

9.2.2 Poisoning Matlab's SVM classifier

Similar to the experiment on Matlab's Logistic Regression classifier, Figures 9.4-9.6 shows the result of using the different sets of poisoning points to poison the `svmtrain` SVM classifier, using setting the `'kernel_function'` option as `'rbf'` (Radial Basis Function) with the *alpha* value as 10 for the Spambase problem and 20 for the Ransomware and MNIST problem. In this experiment, we can observe that the effect from the poisoning attack reduces as the problem grows in the number of dataset features. In the MNIST classification problem, we observe no substantial increase in the classification rate after injecting as much as 15% of poisoning points into the training dataset.

9.2.3 Effect of Different Sets of Poison

We can observe from the graphs in figure 9.1-9.6 the effect of each set of poisoning point generated from different model classification system i.e. our local ADALINE, Logistic Regression, and Multi-Layered Perceptron. Although the observation seems to suggest that the poisoning points are transferable – each set of poison generated by different model classifier has the same effect on the target system –, the result from figure 9.3 suggests that this might not always be true. Figure 9.3 shows that the poisoning point generated by the local Logistic Regression works best to poison the target Logistic Regression classifier `glmfit`. Therefore, the transferability of poisoning points is likely to be dependent on the classification problem and the target classification system. More on this is to be examined in the future.

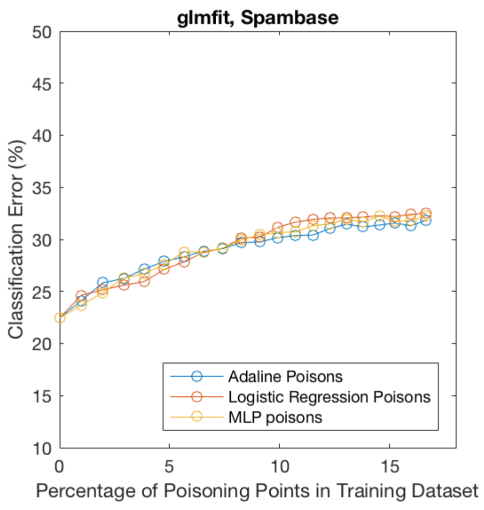


Figure 9.1:

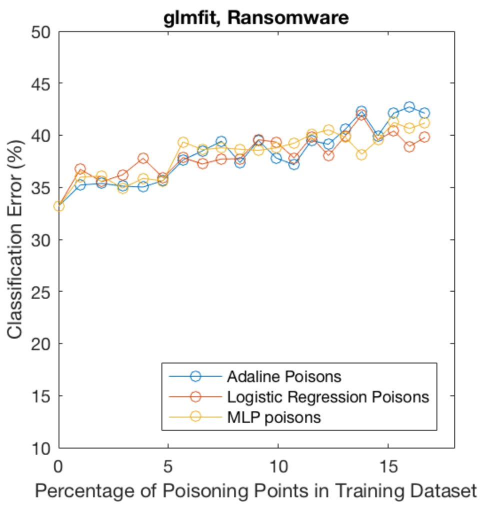


Figure 9.2:

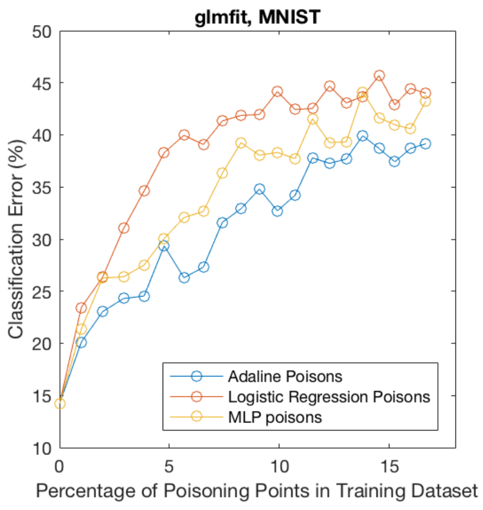


Figure 9.3:

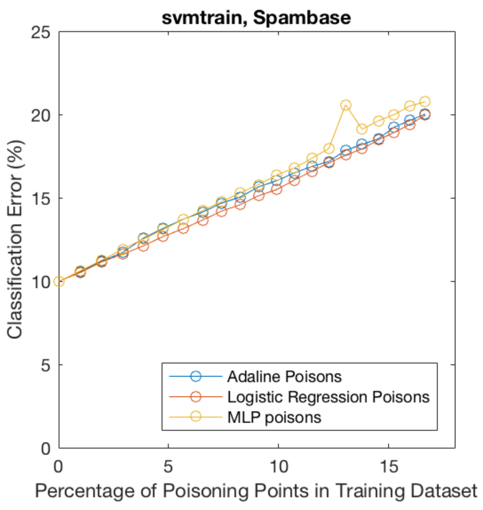


Figure 9.4:

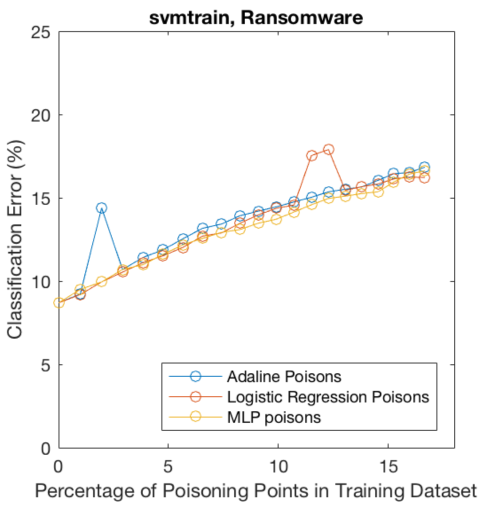


Figure 9.5:

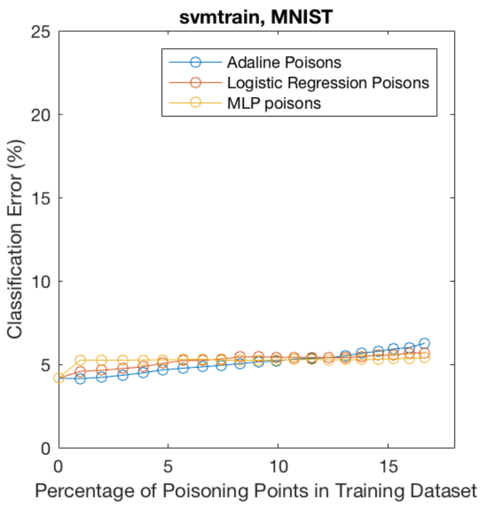


Figure 9.6:

Chapter 10

Conclusion and Future Work

Attacks against machine learning systems are real threats which has been reported before [14] [12] [16] [36]. This work has explored, in detail, ways in which attackers can exploit machine learning systems with the poisoning attack – an attack that is known to be one of the most relevant attacks faced by machine learning systems. With the aim to provide a thorough analysis on the vulnerabilities of well known classifiers – both linear classifiers (ADALINE and Logistic Regression) and the non-linear classifier (Multi-Layered Perceptrons) – against poisoning attacks. Ultimately I have shown that the poisoning attacks can be done in practice, and that such attacks are indeed very harmful to different machine learning classifier systems.

10.1 Work Summary

This section summarises the work I have done for this project.

- I started off by proving and examining the method described in the work Biggio et al. [45] that allows attackers to craft the most harmful poisoning sample by solving a bi-level optimisation problem, using a KKT condition assumption.

- I have then proposed a more efficient and stable method to solve the same bi-level optimisation problem using the conjugate-gradient algorithm. With this method I have carried out experiments to determine how ADALINE and Logistic Regression classifier are affected by the optimal poisoning attack.

- I have explored the novel back-gradient method proposed by L. Muñoz-González [24] to solve the bi-level optimisation problem without having to depend on the KKT stability condition assumption. This has then allowed the bi-level optimisation for a neural network system to be solved, enabling examination of the poisoning attack on neural network systems. With this method I have also carried out the same experiments to determine how ADALINE and Logistic Regression classifier are affected by the optimal poisoning attack.

- Further, as the back-gradient method is known to be very efficient, I have also investigated the time-complexity of the back-gradient method by carrying out time experiments and compare the results with the conjugate-gradient method I proposed.

- I have then explored two attack strategies – greedy and coordinated strategy, and examined, via experiment, how each poisoning strategy performs against each of the three classifier: ADALINE, Logistic Regression, and Multi-Layered Perceptron.

- I have also extended the attack to attacking a multi-class classifier, experimenting with two different attack strategy – targeted and indiscriminated. Work on poisoning multi-class classifiers has not been done before in the literature, hence the results of this experiment have shed light on the issue.

- I have ended the project by carrying out the poisoning attacks in an even more realistic settings i.e. reducing the knowledge of the attacker on the implementation and the learning algorithm of the targeted classifiers. In this experiment I have essentially looked at the transferability of the poisoning samples i.e. investigating whether poisoning samples crafted using one underlying machine learning classifier can be used to poison another (different) classifier.

10.2 Future Work

Parameters Optimisation

As discussed in section 3.4.2, in this work, I have only been estimating the 'right' values of the parameters by looking at the cost graph and by trial-and-error. This is because solving for the optimal value of the parameters is hard, as it requires us to solve another bi-level optimisation problem, similar to solving for optimal hyper-parameter values. This bi-level optimisation problem has been explored by Andrew Ng et al. [9]. I would like to apply that to the work of this project, as successfully solving for the optimal parameters would result in better overall effectiveness of the poisoning attacks. Further, it would also provide an even more fair results to use when comparing the effectiveness of different attack methods or strategies.

Defense Analysis

Various methods such as RONI[19] and Dynamic Threshold Defenses[29] have been introduced to defend against the poisoning attack. I would like to examine their performance in defending against the attack methods we explored in this work. Further, having known how the optimal poisoning attack works, I would like to implement an effective defense mechanism to protect against it.

Constrained Attacks

The optimal poisoning attack method is useful when we want to evaluate the degree of damage that a poisoning attack can cause. However with this method, it is possible that developers of the systems in the real world could use an anomaly detector to separate our poisoning training samples from the legitimate ones. Hence, I would like to explore the constrained poisoning attack method, which is a method that generates poisoning points that are similar enough to the legitimate training sample that they cannot be detected by anomaly detectors.

Poisoning Attack on Recommendation System

So far we have only been looking at classification systems. The attack on the recommended is also very relevant and interesting to look at, as recommendation systems are trained with the users' input. The ability to manipulate the recommendations or suggestions in an application can benefit malicious users in number of ways. I would like to explore the poisoning attack for this problem.

Poisoning Attack on Deep Learning Systems

Many sophisticated machine learning applications nowadays have made use of deep learning architectures. It would be interesting to see if the deep learning systems are also prone to the poisoning attack.

Chapter 11

Appendix

11.1 Parameters for Conjugate-Gradient Method Experiments

11.1.1 Classification Rate Experiment on Spambase dataset

Classifier	Iter	LR	Inner Iter	Inner LR
ADALINE	2000	0.01	2000	0.1
Logistic Regression	1000	0.1	1500	0.5

11.2 Parameters for Back-Gradient Method Experiments

11.2.1 Classification Rate Experiment on Spambase dataset

Parameters-Classifiers	ADALINE	Logistic Regression
Iterations	300	300
Learning Rate	0.2	1
Inner Iterations	2000	1500
Inner Learning Rate	0.1	0.5
Back-Gradient Iterations	100	100
Back-Gradient Learning Rate	0.1	0.1

11.2.2 Classification Rate Experiment on Ransomware dataset

Parameters-Classifiers	ADALINE	Logistic Regression
Iterations	200	300
Learning Rate	1	1
Inner Iterations	2000	1500
Inner Learning Rate	0.05	0.5
Back-Gradient Iterations	80	100
Back-Gradient Learning Rate	0.2	0.1

11.2.3 Classification Rate Experiment on MNIST dataset

Parameters-Classifiers	ADALINE	Logistic Regression
Iterations	200	300
Learning Rate	1	0.5
Inner Iterations	2000	1000
Inner Learning Rate	0.05	0.5
Back-Gradient Iterations	80	80
Back-Gradient Learning Rate	0.02	0.5

11.3 Time Experiment

11.3.1 Conjugate-Gradient Method on Ransomware dataset

Parameter setting for this experiment is the same as in appendix section 11.1.1.

11.3.2 Conjugate-Gradient Method on Ransomware dataset

Classifier	Iter	LR	Inner Iter	Inner LR
ADALINE	200	1	2000	0.5
Logistic Regression	300	1	1500	0.5

11.3.3 Classification Rate Experiment on MNIST dataset

Classifier	Iter	LR	Inner Iter	Inner LR
ADALINE	200	1	2000	0.5
Logistic Regression	300	0.5	1000	0.5

The time experiment parameter settings are the same as that of the classification rate experiments.

11.3.4 Back-Gradient Method

11.3.5 Multi-Layered Perceptron Poisoning Experiment

Parameters-Datasets	Spambase	Ransomware	MNIST
Number of Layers	2	2	2
Neurons in hidden layer	10	10	10
Iterations	300	400	400
Learning Rate	0.2	0.2	0.2
Inner Iterations	1000	400	400
Inner Learning Rate	0.1	0.1	0.1
Back-Gradient Iterations	200	200	200
Back-Gradient Learning Rate	0.1	0.1	0.1

11.4 Parameters for Experiments on Coordinated Attack Strategy

11.4.1 Classification Rate Experiment on Spambase Dataset

Parameters-Classifier	ADALINE	Logistic Regression	MLP
Number of Layers	n/a	n/a	2
Neurons in hidden layer	n/a	n/a	10
Iterations	300	300	300
Learning Rate	0.5	0.3	0.2
Inner Iterations	2000	1500	1000
Inner Learning Rate	0.1	0.5	0.1
Back-Gradient Iterations	100	100	200
Back-Gradient Learning Rate	0.1	0.1	0.1

11.4.2 Classification Rate Experiment on Ransomware Dataset

Parameters-Classifier	ADALINE	Logistic Regression	MLP
Number of Layers	n/a	n/a	2
Neurons in hidden layer	n/a	n/a	10
Iterations	200	300	300
Learning Rate	0.2	0.1	0.2
Inner Iterations	2000	1500	1000
Inner Learning Rate	0.05	0.5	0.1
Back-Gradient Iterations	80	100	200
Back-Gradient Learning Rate	0.02	0.1	0.01

11.4.3 Classification Rate Experiment on MNIST Dataset

Parameters-Classifier	ADALINE	Logistic Regression	MLP
Number of Layers	n/a	n/a	2
Neurons in hidden layer	n/a	n/a	10
Iterations	300	300	300
Learning Rate	0.2	0.3	0.1
Inner Iterations	2000	1000	1000
Inner Learning Rate	0.05	0.5	0.1
Back-Gradient Iterations	80	80	1000
Back-Gradient Learning Rate	0.02	0.5	0.1

11.5 Parameters for Experiments on Multi-Class Logistic Regression Classifier

11.5.1 Targeted Strategy

Parameter	Value
Iterations	20
Learning Rate	10
Inner Iterations	400
Inner Learning Rate	0.5
Back-Gradient Iterations	60
Back-Gradient Learning Rate	0.2

11.5.2 Indiscriminate Strategy

Parameter	Value
Iterations	20
Learning Rate	10
Inner Iterations	400
Inner Learning Rate	0.5
Back-Gradient Iterations	60
Back-Gradient Learning Rate	0.2

Bibliography

[1] Abdeslam, D. O., Wira, P., Mercklé, J., Flieller, D., and Chapuis, Y.-A. A unified artificial neural network architecture for active power filters. *IEEE Transactions on Industrial Electronics 54*, 1 (2007), 61–76.

[2] Andrea Paudice, Luis Muñoz-González, A. G., and Lupu, E. C. A critical survey on poisoning attacks against learning systems.

[3] Baldi, P., and Brunak, S. *Bioinformatics: the machine learning approach*. MIT press, 2001.

[4] Barreno, M., Nelson, B., Joseph, A. D., and Tygar, J. The security of machine learning. *Machine Learning 81*, 2 (2010), 121–148.

[5] Bergstra, J., and Bengio, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research 13*, Feb (2012), 281–305.

[6] Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems* (2011), pp. 2546–2554.

[7] Burges, C. J. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery 2*, 2 (1998), 121–167.

[8] C. Blake, C. M. Uci repository of machine learning database ,https://www.ics,uci.edu/ mlearn/mlrepository.html, 1998.

[9] Chung B. Do, Chuan-Sheng Foo, A. Y. N. Efficient multiple hyperparameter learning for log-linear models.

[10] Daniele Sgandurra, Luis Muñoz-González, R. M. E. C. L. Automated dynamic analysis of ransomware: Benefits, limitations and use for detection, 2016.

[11] Davidian, D. Feed-forward neural network, Aug. 1 1995. US Patent 5,438,646.

[12] Dennis Batchelder, H. J. Immunity from antimalware automation attacks.

[13] Friedman, J., Hastie, T., and Tibshirani, R. *The elements of statistical learning*, vol. 1. Springer series in statistics Springer, Berlin, 2001.

[14] Garcia, M. How to keep ai from turning into a racist monster.

[15] Gardner, M. W., and Dorling, S. Artificial neural networks (the multi-layer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment 32*, 14 (1998), 2627–2636.

[16] Gendreau, H. The internet made fake news a thing then made it nothing.

[17] Guyon, I., and Elisseeff, A. An introduction to variable and feature selection. *Journal of machine learning research 3*, Mar (2003), 1157–1182.

[18] Hestenes, M. R., and Stiefel, E. *Methods of conjugate gradients for solving linear systems*, vol. 49. NBS, 1952.

[19] Huang, L., Joseph, A. D., Nelson, B., Rubinstein, B. I., and Tygar, J. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence* (2011), ACM, pp. 43–58.

[20] Kawaguchi, K. University of texas, ece online material.

[21] Kira, K., and Rendell, L. A. A practical approach to feature selection. In *Proceedings of the ninth international workshop on Machine learning* (1992), pp. 249–256.

[22] Krantz, S. G., and Parks, H. R. *The implicit function theorem: history, theory, and applications*. Springer Science & Business Media, 2012.

[23] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[24] L. Muñoz-González, A. Paudice, A. D. B. B. F. R. E. L. Poisoning neural networks with back-gradient optimization". international conference on machine learning (submitted). ICML.

[25] LeCun, Y., Jackel, L., Bottou, L., Cortes, C., Denker, J. S., Drucker, H., Guyon, I., Muller, U., Sackinger, E., Simard, P., et al. Learning algorithms for classification: A comparison on handwritten digit recognition. *Neural networks: the statistical mechanics perspective 261* (1995), 276.

[26] Lison, P. "an introduction to machine learning, 2015.

[27] Lyon, A. Why are normal distributions normal? *The British Journal for the Philosophy of Science 65*, 3 (2013), 621–649.

[28] Mei, S., and Zhu, X. Using machine teaching to identify optimal training-set attacks on machine learners. In *AAAI* (2015), pp. 2871–2877.

[29] Nelson, B., Barreno, M., Chi, F. J., Joseph, A. D., Rubinstein, B. I., Saini, U., Sutton, C. A., Tygar, J. D., and Xia, K. Exploiting machine learning to subvert your spam filter. *LEET 8* (2008), 1–9.

[30] Ng, A. Coursera: Machine learning online course.

[31] Papernot, N., McDaniel, P., Sinha, A., and Wellman, M. Towards the science of security and privacy in machine learning. *arXiv preprint arXiv:1611.03814* (2016).

[32] Pazzani, M., and Billsus, D. Content-based recommendation systems. *The adaptive web* (2007), 325–341.

[33] Pearlmutter, B. A. Fast exact multiplication by the hessian. *Neural computation 6*, 1 (1994), 147–160.

[34] R. A. Fisher, M. M. Uci repository of machine learning database ,https://archive.ics.uci.edu/ml/datasets/iris, 1988.

[35] Robinson, G. A statistical approach to the spam problem. *Linux journal 2003*, 107 (2003), 3.

[36] Smutz, C., and Stavrou, A. Malicious pdf detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), ACM, pp. 239–248.

[37] Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the importance of initialization and momentum in deep learning. In *International conference on machine learning* (2013), pp. 1139–1147.

[38] Tino, P. Machine learning and computational finance.

[39] Tsai, J. J., and Philip, S. Y. *Machine learning in cyber trust: security, privacy, and reliability.* Springer Science & Business Media, 2009.

[40] Tsai, J. J., and Philip, S. Y. *Machine learning in cyber trust: security, privacy, and reliability.* Springer Science & Business Media, 2009.

[41] University, S. Standford deep learning online tutorial.

[42] Widrow, B., and Lehr, M. A. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE 78*, 9 (1990), 1415–1442.

[43] WikiPedia. The conjugate gradient.

[44] Wing, J. M. Computational thinking. *Communications of the ACM 49*, 3 (2006), 33–35.

[45] Xiao, H., Biggio, B., Brown, G., Fumera, G., Eckert, C., and Roli, F. Is feature selection secure against training data poisoning? In *ICML* (2015), pp. 1689–1698.