Final Report

CM3203 - One Semester Individual Project - 40 Credits

Adversarial Reasoning in Machine Learning for Natural Language Processing: the case of spam emails

Author - Hasna Ahmed Al Jufaili

Supervisor - Dr.Federico Cerutti

Moderator - Dr.Philipp Reinecke

A Final Year Project Submitted for the Degree of Bachelors of Science

Department of Computer Science with Security and Forensics

Cardiff University

2018

Abstract

Today, the most successful spam filters are built upon the statistical theories of Machine Learning algorithms, since they are easy to construct, train and have shown great performance in the field of spam filtering. However, security issues arise from the Machine Learning model's adaptability in the presence of an adversary that can subvert the Machine Learning model and manipulate the testing samples during the model test time, which poses potential security issues. The purpose of this project is to investigate the performances of spam filters built upon Machine Learning in the presence of an adversary and examine the possibility of identifying a Black Box model underlying algorithm using adversarial examples designed for other models. A software system has been implemented to model the combat between spam filters that are based on the most popular Machine Learning Classification algorithms and possible adversarial attacks using Python programming language. Spam filters built upon Machine learning algorithms were found to be vulnerable to adversarial attacks. In addition, identifying the underlying algorithm of a Black Box model was found to be possible. Thus, adversarial attacks on applications based on Machine Learning raise major security issues that need to be considered.

Acknowledgments

I would like to take this opportunity to express my gratitude to the individuals who have supported me throughout the whole project. A massive thank you to my supervisor Dr.Federico Cerutti for his continued enthusiasm and invaluable support throughout the whole project and I have no doubt that without his support, the project would have been far less of a success. Another thank you to my family and friends, who inspire me with confidence and beyond and I could not have done this project and complete my Bachelor's degree without their support, which I will forever cherish.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Preface

Nowadays, emails became a major part of individuals daily life as a vital communication tool. Every second, billions of emails are sent around including spam emails, where spam emails are irrelevant or unsolicited emails that contain links that might look familiar to the user but leads to phishing web sites or malware hosted sites and contain malwares as scripts or executable file attachments that may harm the user. According to IBM Security, spam emails are considered as a primary tool in each attacker's toolkit [8]. In addition, research done by IBM security showed that the number of spam emails is increasing rapidly, and the volume increased by 4x (Fourfold) in 2016 [8]. Therefore, to protect the users from the threat of spam emails, email spam filters and detectors are used and provided by email services and several software's to keep the spam emails out of the user's inbox. These filters, use statistical Machine Learning Classification techniques and algorithms to produce intelligent decisions on the Classification of the data in the emails to decide whether an email is spam or not [2]. These statistical algorithms are implemented into a Machine Learning model that adapt the application to the changes in data by learning and training constantly on given datasets, which then produces a set of patterns and rules that will be followed to classify future data [15]. However, security issues arise from the Machine Learning model's adaptability in the presence of an adversary that can fool the Machine Learning model by manipulating the input data during the test time of the model [15].

1.2 Project Aims and Scope

The main aims of this project are to investigate the performances of spam filters built upon different Machine Learning algorithms in the presence of an adversary and examine the possibility of identifying a Black Box model underlying algorithm using adversarial examples designed for other models.

The project scope focuses on implementing a software system that allows the users to examine the performances of different Machine Learning Classification algorithms acting as spam filters against several adversarial attacks.

1.3 Intended Audience

The intended audience and beneficiaries from this project are the individuals who are interested or doing research in the field of security of applications built on Machine Learning algorithms and more specifically the case of spam filtering.

1.4 Report Structure

This paper is organised in the following way, **Chapter 2** presents an initial background of Machine Learning, Machine Learning approach to Spam Filtering, measures for evaluating Machine Learning classifiers and Adversarial Machine Learning, **Chapter 3** discusses the requirements and design of the software system, **Chapter 4** focuses on the implementation of the software system down to the code level, **Chapter 5** presents the test cases that were undertaken to test the software system, **Chapter 6** focuses on the results that were obtained from the developed software system, **Chapter 7** focuses on the potential future work that could be undertaken to improve the project, **Chapter 8** concludes the project and summarises the main findings and **Chapter 9** reflects on the learning obtained from undertaking this project.

Chapter 2

Background Research

2.1 Machine Learning

Nowadays, **Machine Learning** has become a popular tool in any task that requires extraction of information from large datasets. The term Machine Learning means the acquisition of structural patterns from examples for future prediction, explanation and understanding purposes [20]. It provides the technical basis of **Data mining**, which is the extraction of implicit, previously unknown, and potentially useful information from data [20]. Some applications that utilises Machine Learning are, credit card fraud detection, spam filters and digital cameras face detectors [7]. These applications are based on the theory of statistical algorithms in building learning models to make interference from a sample. In addition, Machine Learning approaches provide an automated, adaptive approach that extracts knowledge from supplied training datasets using a Machine Learning algorithm such as **Classification**, to utilise the information obtained in the categorisation of previously unseen data as seen in Figure 2.1 [6], where Classification is a form of data analysis that extracts models categorising significant data classes. These models are referred to as **Classifiers**, which predict categorical (unordered, discrete) class labels for given data [7]. For example, having a table of data that has different individuals information that includes their name, age, income and a categorical class loan decision as seen in Figure 2.2, a classifier will be able to predict the categorical class loan decision for an unseen individual after extracting some rules using a classification algorithm from the data in the table. Several classification techniques have been used by researchers in the fields of Machine Learning, statistics and pattern recognition for applications including fraud detection, medical diagnosis, performance prediction and

target marketing [7].

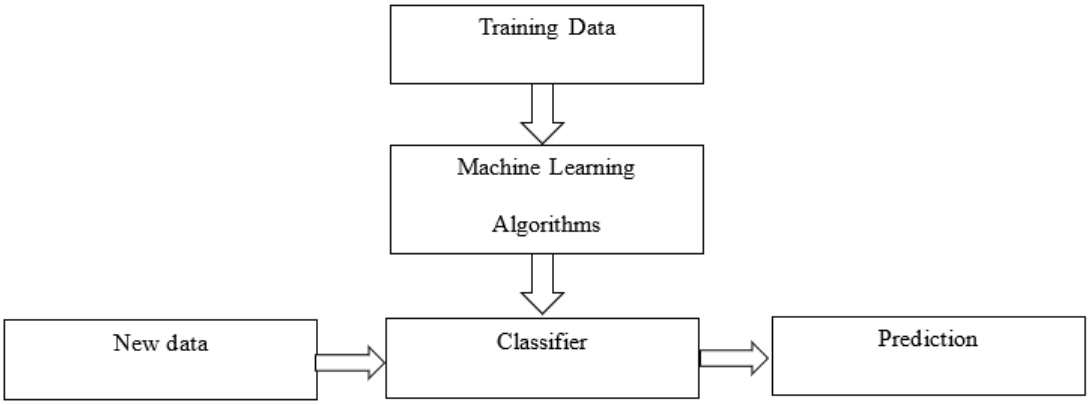

Figure 2.1: Machine Learning Flow

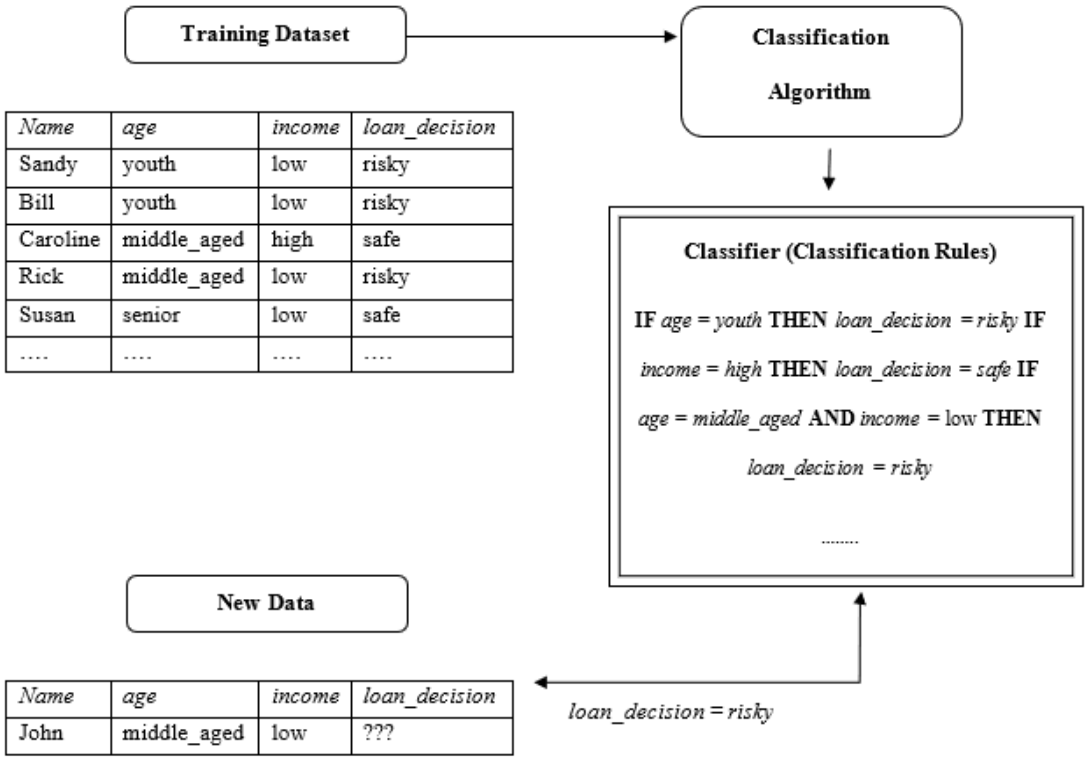

Figure 2.2: Machine Learning using Classification

In addition, classification process consists of two main steps, learning step (training phase), where a classifier is built using a classification algorithm by analysing and learning from an available training set and their class labels and classification step, where previously unseen

data are classified and categorised as seen in Figure 2.2 [7]. Several Machine Learning classifiers exist, some of the most popular used Machine Learning classifiers will be discussed in the upcoming sections.

2.1.1 Supervised VS Unsupervised Machine Learning

There exists several types of Machine Learning, **Supervised** and **Unsupervised** Machine Learning are mostly used [10]. Supervised Machine Learning is learning in presence of training dataset that is already labelled, for example, a set of emails that have already been labelled as either spam or ham (ham represents legitimate and good emails) [17]. Therefore, the obtained expertise and knowledge from the given training dataset is used to predict the label of unseen data, which is illustrated in Figure 2.3. Therefore, Classification is a typical example of Supervised Machine Learning [1].

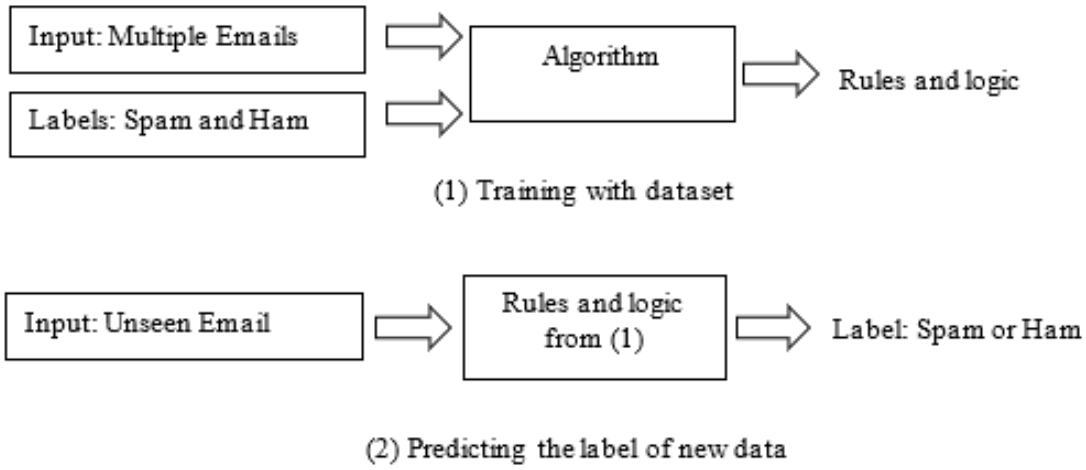

Figure 2.3: Supervised Machine Learning

In Unsupervised Machine Learning, a training dataset is provided but lacks from labels, for example, a set of emails that are not labelled if they are either spam or ham. Thus, there exists no distinction between training and unseen test data and the machine will have to extract and understand patterns from the given data as shown in Figure 2.4 [17]. An example of Unsupervised Machine Learning is **Clustering**, which divides the data in a dataset into subsets of similar objects [17]. For the purposes of this project Supervised Machine Learning

will be utilised using Classification methods.

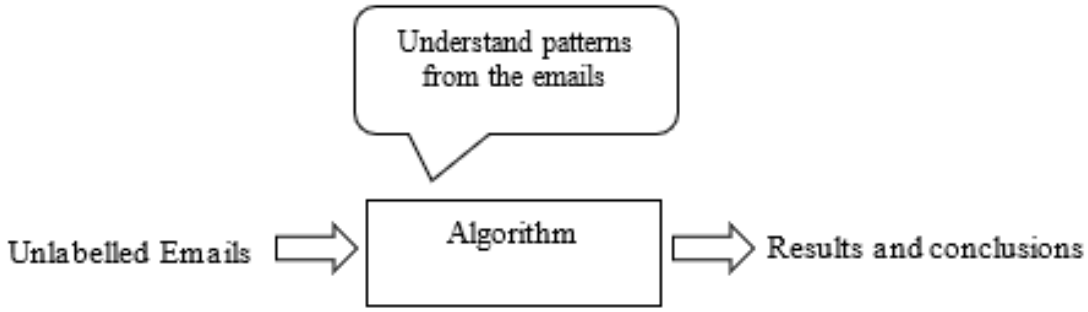

Figure 2.4: Unsupervised Machine Learning

2.2 Machine Learning Approach to E-mail Spam filtering

Email spam filtering is considered as a binary classification task, where spam emails are treated as positive (+) instances, and ham emails (legitimate) as negative (-) instances [2]. Therefore, Machine Learning techniques and statistical approaches are used to build classifiers that will filter spam emails from a user mail stream. As suggested by Bhowmick & Hazrika, some of the most popular Machine Learning classifiers that are used for spam filtering are Naïve Bayes, Support Vector Machines and Decision Trees [2]. Each one of the classifiers mentioned above is further discussed in the following sections.

2.2.1 Naïve Bayes

In Machine Learning, **Naïve Bayes** classifier is one of the most popular statistical spam filters [2] and it is based on **Bayes' theorem** with an assumption of independency among the input features (document words) [14]. Thus, it assumes that the occurrence of a certain feature in a class is unrelated to the occurrence of other features. In the spam filter scenario, each email is represented as a vector $\vec{x} =< x_i, ..., x_n >$ with n features (each feature represents a word that appear in the corpus) and the email could be categorised into two different classes, either spam or ham $\{C_s, C_h\}$, where C_s represents the spam class and C_h

represents the ham class [14]. Using the Bayes' theorem, the probability of an email with vector $\vec{x} =< x_i, ..., x_n >$ to have a class C_i that is $\in \{C_s, C_h\}$ can be calculated as follows,

$$P(C_i|\vec{x})_{C_i \in \{C_s, C_h\}} = \frac{P(C_i).P(\vec{x}|C_i)}{P(\vec{x})},$$

where $P(C_i)$ is the probability of observing the class C_i in the training set, $P(\vec{x}|C_i)$ is the probability of having a message classified as C_i is represented by $\vec{x}$ and $P(\vec{x})$ is the probability of observing the message $\vec{x}$ [2]. Since Naïve Bayes assumes independence among the features of $\vec{x}$, then $P(\vec{x}|C_i)$ could be changed into the following form [1],

$$P(\vec{x}|C_i) = \prod_{i=1}^{n} P(x_i|C_i)$$

Therefore, the probability of an email with vector $\vec{x} =< x_i, ..., x_n >$ to have a class that is $\in \{C_s, C_h\}$ can now be calculated as follows,

$$P(C_i|\vec{x})_{C_i \in \{C_s, C_h\}} = \frac{P(C_i) \prod_{i=1}^{n} P(x_i|C_i)}{P(\vec{x})},$$

the Naïve Bayes optimal classifier that will be used to classify unseen emails can be then interpreted as [9],

$$\hat{C}_i = \underset{C_i \in \{C_s, C_h\}}{\operatorname{argmax}} P(C_i) \prod_{i=1}^{n} P(x_i|C_i),$$

where $\hat{C}_i$ represents the class label of the unseen email that has the highest probability value and $P(x_i|C_i)$ calculation depends on the type or form of Naïve Bayes classifier. Thus, Naïve Bayes represents each class with a probabilistic summary and classifies each email with the most likely class it finds [5].

2.2.2 Decision Trees

A **Decision Tree** is a hierarchical model used in Supervised Machine Learning, since it yields to a sequence of intermediate decisions, which lead to a final decision and has been seen to give good results and being efficient for classification tasks [10]. A Decision Tree

is represented as a flowchart-like tree structure that consists of nodes, branches and leaf nodes [7]. Each node represents a test on a feature, each branch represents an outcome of a test at a feature node and the leaf nodes represent classes. Therefore, Decision Trees can be easily interpreted as classification rules [7]. To construct a Decision tree, Decision Tree induction algorithms are used. One of the most popular algorithms is known as ID3 with an extended version called C4.5. These algorithms are intended for classification tasks, they construct a decision tree from a set of given data by choosing the set of features that are mostly useful to a certain classification task and represents them as the nodes to partition the data into different classes [7]. Therefore, the features that do not appear are considered irrelevant [7]. A simple Decision Tree represent the spam filtering scenario is presented in Figure 2.5. The root node as it can be seen, checks the occurrence frequency of the feature "Free", if it occurred in the email more than two times, then it moves to the next internal node, otherwise the email is classified as ham. Next, it checks the occurrence frequency of the feature "Money", if it is more than two, the email is therefore classified as spam and ham otherwise.

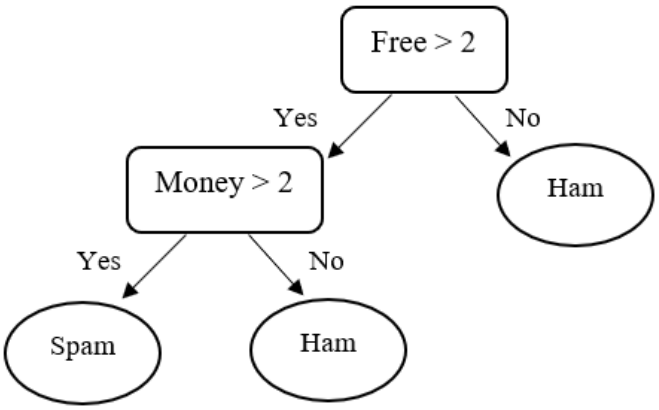

Figure 2.5: Simple Decision Tree

2.2.3 Support Vector Machines

A **Support Vector Machine** is a Supervised learning method that has been extensively used for classification tasks yielding a great success rate [2]. For the two-class (e.g spam and ham) separable training set scenario, the main task of a Support Vector Machine is to

find and create an optimal hyperplane (boundary) that divides the data into two different classes (spam and ham) by maximising the distance called margin between the separating hyperplane and the closest data points [10]. The reason behind choosing the hyperplane with the largest margin is because it makes the classification correct for testing data that are near, but not identical to the training data [10]. A simple example is shown in Figure 2.6, where the samples belong either to class spam represented by (+) or ham represented by (-). The separating line (hyperplane) defines a boundary on the right side of which all emails are spam, and to the left of which all emails are ham. Any new unseen email falling to the right will be classified as spam and otherwise it will be classified as ham if it falls to the left of the line.

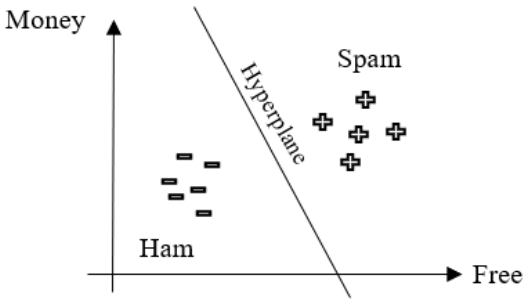

Figure 2.6: Support Vector Machine

2.3 Evaluating what has been learned

Evaluation is the key to success in Machine Learning and Data Mining [20]. To determine which classification algorithm is suitable for a particular classification problem or evaluate how well different algorithms work and compare one with another, systematic evaluation methods are used on the test set [20]. The reason why evaluation is done on a test set that is different than the training set is to avoid model **Overfitting**. If no data splitting was done, same data will be used for training and testing, which will result in the model repeating the labels of the samples that were previously seen while training in the testing phase, which will result in having a perfect accuracy score for predicting the testing set but will fail in predicting unseen data [20]. In the two-class case with classes spam or ham, yes or no, and

so on, each prediction can have four different outcomes, True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN) where these outcomes are defined as follows,

- TP: correct classification of a positive class
- TN: correct classification of a negative class
- FP: incorrect classification of a positive class that is actually negative
- FN: incorrect classification of a negative class that is actually positive

Furthermore, the total result of the outcomes is usually presented in a Confusion matrix of two dimensions as seen in Figure 2.7, where each class (Positive and Negative) is represented by a single row for the actual class and a single column for the predicted class.

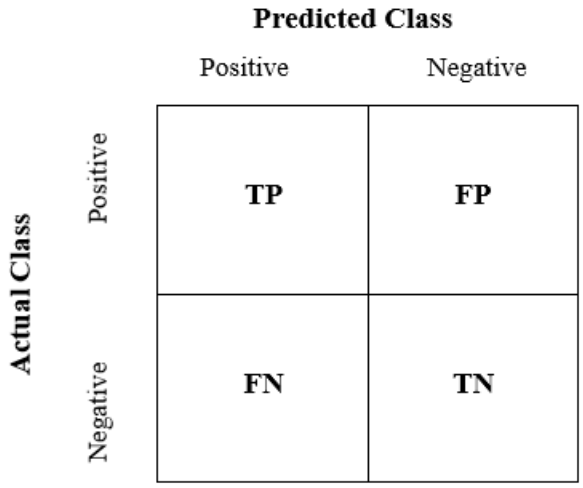

Figure 2.7: Confusion Matrix

Each matrix element then represents the total number of the test instances for which is the actual class is the row and the predicted class is the column [20]. Thus, a good performance indicator from the Confusion matrix would be having large numbers in the diagonal elements and small numbers, ideally zero in the off-diagonal elements. In addition, these outcomes that are presented in the Confusion matrix are used by multiple evaluation metrics such as, Accuracy, Recall and Precision to evaluate a classifier's performance. Each evaluation metric

simply summarises the confusion matrix differently as seen in the following list [20],

- Accuracy: Accuracy is the overall success rate that is calculated by dividing the number of correct classifications (TP and FP) by the total number of the classifications, the best value is 1 and the worst is 0 [20].
- Recall: Recall is the ability of the classifier to find all the positive instances and is calculated by dividing the number of correct positive classifications (TP) by the (TP+FN), the best value is 1 and the worst is 0 [20].
- Precision: Precision is the ability of the classifier to not label a negative class instance as positive and is calculated by dividing the number of correct positive classifications (TP) by the (TP+FP), the best value is 1 and the worst is 0 [20].

2.4 Adversarial Machine Learning

Adversarial Machine Learning is the study of the vulnerabilities of Machine Learning algorithms in a presence of an adversary (opponent) that targets a Machine Learning model during the testing phase when making predictions or the training phase of the model parameters [4]. In the testing phase scenario, the adversary uses crafted malicious inputs (adversarial examples) that are designed to be miss-classified by the Machine Learning model [4]. It has been seen that several Machine Learning models are often vulnerable to adversarial changes in their input, which causes incorrect classification [12]. In Addition, it has been seen that adversarial examples designed for a specific model have a transferability property (able to be miss-classified by an another Machine Learning model without having the knowledge of the underlying model), where one adversarial example that has been designed to be miss-classified by a model denoted as "Model1" is also often miss-classified by another unknown model denoted by "Model2" [12]. This allows the attackers to mount the adversaries on Black box models. Therefore, Adversarial examples pose major security risks for applications based on Machine Learning.

2.5 Python Machine Learning libraries

Since python programming language will be used in this project for implementation purposes, research on libraries supporting Machine Learning had to be taken. Python provides several open source Machine Learning libraries, such as, Scikit-learn library, Theano and Keras. Scikit-learn library consists of concise and consistent interface to the common Machine Learning algorithms, which makes it easier to achieve Machine Learning in production systems [16]. The library contains a good documentation and quality code and combines the ease of use and high performance. Theano is a python package that is similar to NumPy library (A library that provides useful features and operations that can be used on n-arrays and matrices in python) as it defines multi-dimensional arrays and provides math operations and expressions using NumPy syntax [18]. In addition, the library also optimises the use of GPU and CPU, making the performance of data-intensive computation even faster [18]. Keras is an open source library that is dedicated more for a subfield of Machine Learning called **Deep Learning** [11]. For the purposes of this project, Scikit-learn will be used over Theano and Keras since the data that will be analysed is not large and because Scikit-learn provides interfaces for the Machine Learning algorithms, which will help in developing the software system faster by reusing them.

2.6 Existing Solutions

An existing solution that is relevant to the area of this project is Adversarial Machine Learning Library [1] (adlib) that is developed by the Adversarial Machine Learning Group within the Vanderbilt Computational Economics Research Lab led by Dr. Yevgeniy Vorobeychik. This solution provides a library that is written in Python programming language that models the combats between spam email attackers and robust spam filters, using the adversarial machine learning methods. This library includes a data processing unit, several attacks and a simple learner wrap up as a spam filter and several robust learning algorithms. One of the differences between this solution and the solution to be developed is that the developed

[1] Available from https://github.com/vu-aml/adlib/

solution will be implemented as a command line interface that allows user interactions. In addition, the interface will allow users to choose between the most popular classifiers (Naïve Bayes, Decision Tree and Support Vector Machine) when adlib supports only Support Vector Machine. Another difference is that the developed solution will have the option to view a summary of all classifiers performances evaluation before and after mounting the attacks. Finally, the developed solution will allow creating a Black Box model that can be attacked using existing adversarial examples designed for other models.

2.7 Research Questions

Aims

The main aims of this project are to investigate the performances of spam filters built upon different Machine Learning algorithms in the presence of an adversary and examine the possibility of identifying a Black Box model underlying algorithm using adversarial examples designed for other models.

Research questions

In order to demonstrate the achievement of the stated aims, this project will implement the most popular Machine Learning Classification algorithms and a Black Box model, identify a range of adversarial attacks, implement the adversarial attacks, implement suitable evaluation metrics to evaluate the classifiers and then combine all the partial implementations into one usable software system.

Chapter 3

Specification and Design

In order for the project to be successful, it is important that sufficient time is spent on the specification and design of the software system. Therefore, This chapter of the report presents a clear picture of the software system to be developed. It discusses the software system requirements and initial design. In addition, it mentions the development strategy that is followed to produce the software.

3.1 Software Requirement Specification

As with any software system a Software Requirement Specification is produced from the client requirements. The Software Requirement Specification for this project includes several functional and nonfunctional requirements concerning the technical implementation of the software system. These requirements are useful as they allow the evaluation of the success of the final product.

3.1.1 Functional Requirements

The most crucial functional requirements are defined in the following list,

- The system shall take data from the user input as raw email files and their labels for training and testing classifiers.
  - Raw email files will be needed for training and testing the classifiers. Therefore, the system will get the data from the user input by specifying the directory that includes the data.

- The system shall pre-process the given raw emails.
  - Once the raw emails are provided by the user, the system will pre-process them by using Natural Language Processing functions (Tokenisation, Stemming and removal of stop words) to improve the accuracy and simplify extracting the feature vector.
- The system shall provide different classification algorithms to create a Machine Learning model (Classifier).
  - The system will have three different classification algorithms to create a classifier, Naïve Bayes, Support Vector Machines and Decision trees. This is because classification algorithms differ in the following criteria which leads to different results,
    - Predictive Accuracy: this refers to the ability of the model to correctly predict the class label of new or previously unseen data
    - Speed: this refers to the computation costs involved in generating and using the model
    - Robustness: this is the ability of the model to make correct predictions given noisy data or data with missing values
    - Scalability: this refers to the ability to construct the model efficiently given large amount of data.
- The system shall provide an option to create a Black Box model with unknown underlying algorithm.
  - This model will be used to examine the possibility of identifying the underlying algorithm of Black Box models.
- The system shall train the chosen classification algorithm (including the Black Box model) using provided training emails to create a classifier
  - Upon receiving pre-processed training emails, the system will train the chosen classification algorithm to create the classifier.

- The system shall allow the user to attack the created classifier (including the Black Box model) using different attacks.
  - The system will give the user an option to choose an attack to attack the created classifier during the testing phase.
- The system shall allow the user to save an adversarial set that is resulted from attacking their chosen classifier.
  - Existing adversarial sets are needed for attacking Black Box model and also to make it easier to use the same attack again.
- The system shall allow the user to attack trained classifiers using existing saved adversarial sets that were designed for other models.
  - The system will give the user an option load an adversarial set and attack their chosen classifier using it.
- The system must be able to allow a classifier to predict the class/label of previously unseen emails.
  - Given unseen emails that includes no label, the system will allow a trained classifier to predict the label of the unseen emails.
- The system shall allow validating and evaluating trained classifiers before and after attacks.
  - Having a trained classifier, the system will present the results from calculating the Accuracy, Precision and Recall scores of the Classifier before and after attacks.

The optional and desirable functional requirements are defined in the following list,

- The system could allow saving a trained classifier.
- The system could allow loading a trained classifier.
- The system could provide an option that shows a summary of evaluation for all the classifiers.

- The system could provide an option that shows a summary of evaluation for all the classifiers against all possible attacks.

3.1.2 Non-Functional Requirements

The non-functional requirements are defined in the following list,

- Reliability
  - The system will have no errors and will be available to use all the time.
- Usability
  - The system will be implemented as simple command line software that will be easy to use and adjust to.
- Speed
  - Pre-processing the datasets, creating the classifiers, testing and validating and evaluating the classifiers should be fast.
- Size
  - The size of the system will not exceed 1000 megabytes and will have small impact on computer memory.
- Re-usability
  - The implementation of the system will be broken into sub packages with multiple modules that can be reusable in other projects or systems.

3.2 System Architecture

The system will be implemented as a main package called ADML (Adversarial Machine Learning) that will include three sub packages, DataReader, Classifiers and Attacks as seen in Figure 3.1. This utilisation of dividing the system into packages allows easier modification and future additions to the system. DataReader sub package is considered as the most crucial to the functionality of the software system. It will be responsible of preparing and pre-processing the raw emails in order to create feature vectors that will be used by the classifiers and the attacks. Classifiers sub package will be responsible of implementing Naïve Bayes, Decision Tree and Support Vector Machine classifiers. Finally Attacks sub package will include the implementation of the adversarial attacks.

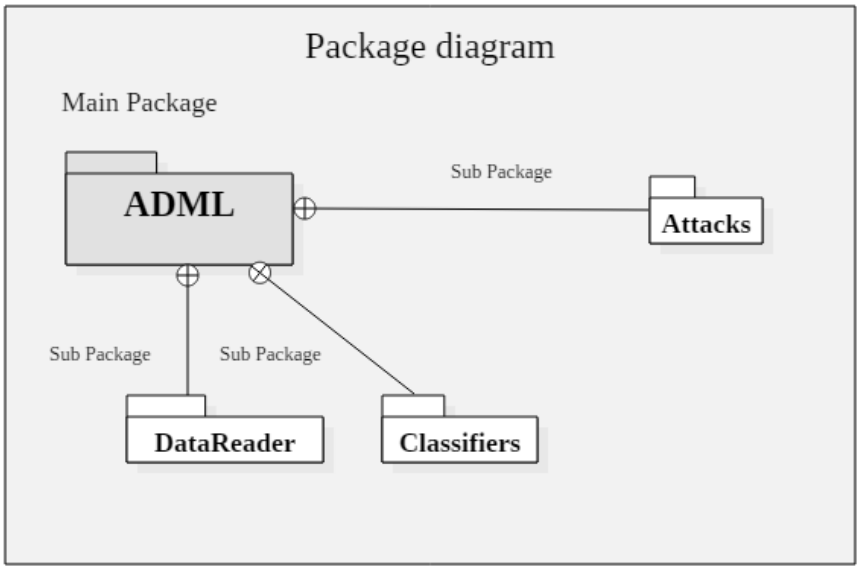

Figure 3.1: Package Diagram

3.3 Design

To better understand the software system to be developed, Unified Modelling Language (UML) has been utilised. UML helps to visualise a software system as it is intended to be and gives a template that guides the developer in constructing the software system. Thus, two different diagrams provided from UML have been created and presented below.

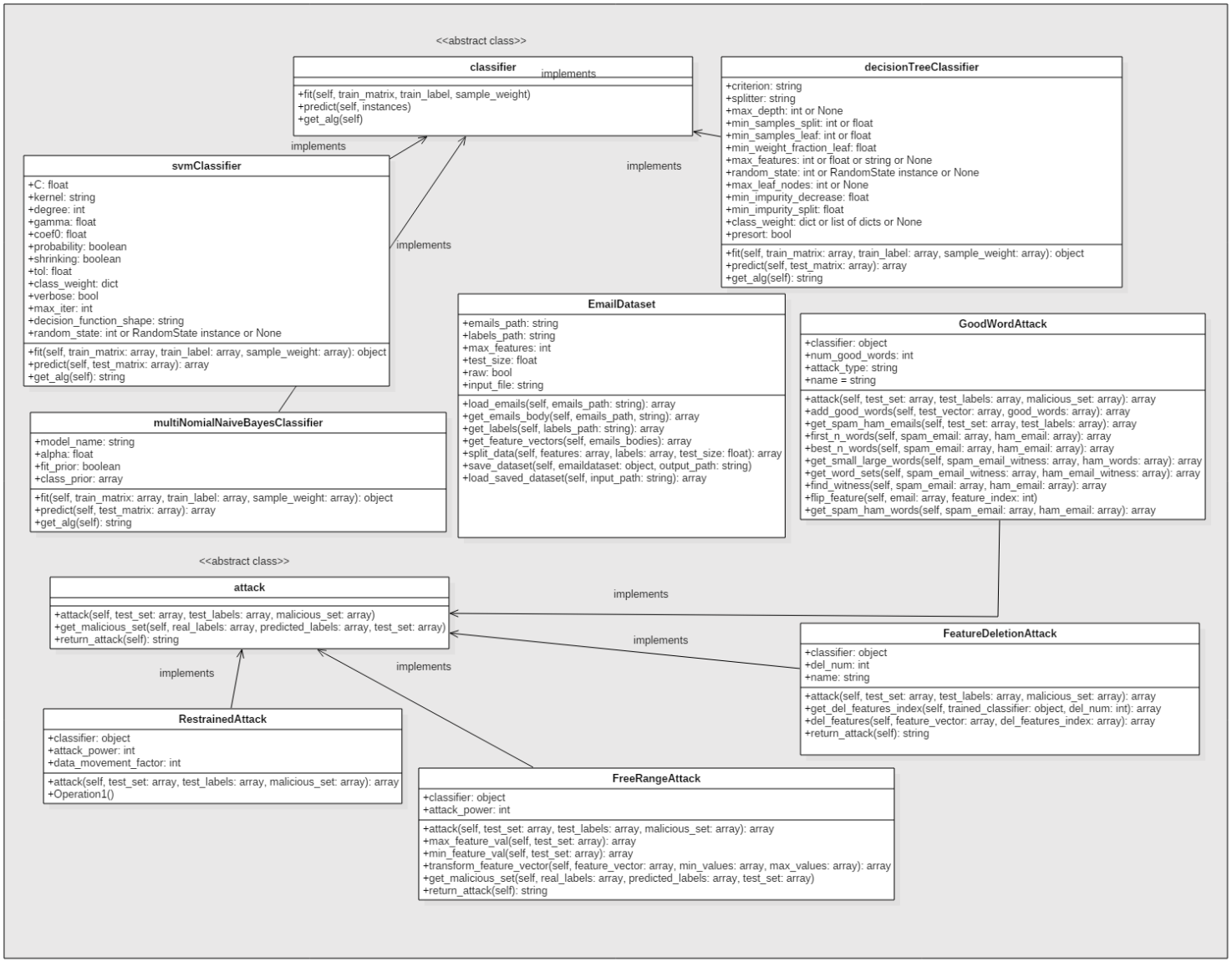

Figure 3.2: Class Diagram

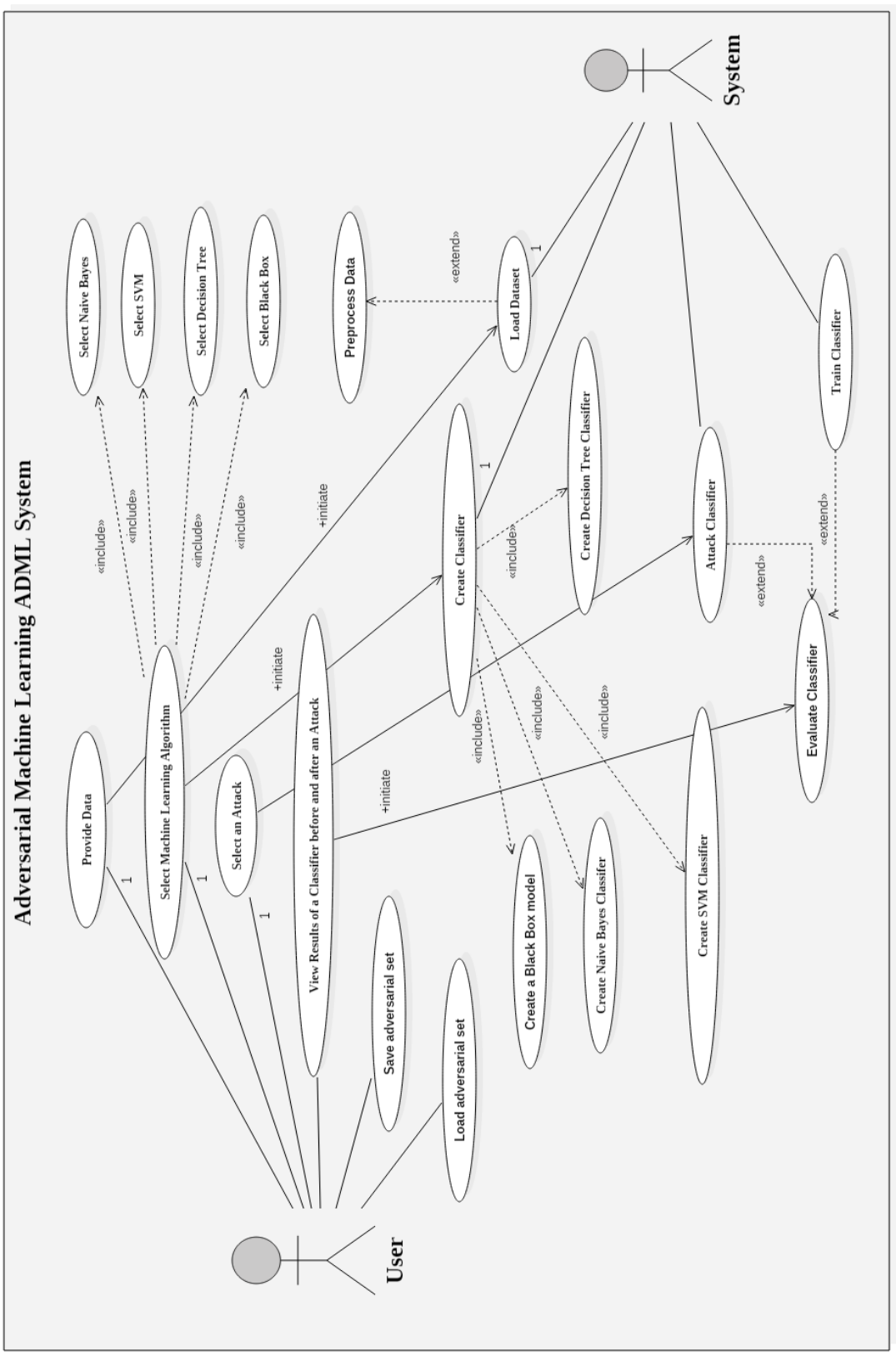

Figure 3.3: Use Case Diagram

Class Diagram

Figure 3.2, shows the internal structure of the software system and the interaction of different classes. The soft copy of the figure is attached as an external file in the submission to provide better vision. The most important classes are described briefly in the following list,

- classifier: This class represents an abstract class for all classifiers. This class is useful because it defines the necessary operations for any classification model and will allow the addition of classifiers with ease at any time during the project.
- EmailDataset: This class represents a dataset, which loads data from raw email files and labels and then pre-processes the data to create feature vectors. This class is important because it allows loading the data into the system and provides the suitable data represented by feature vectors to be used in the software system by other classes.
- attack: This class represents an abstract class for all attacks. This class is useful because it defines the necessary operations for any attack and will allow the addition of attacks with ease at any time during the project.

The classes svmClassifier, decisionTreeClassifier and multiNomialNaiveBayesClassifier have been created to represent the chosen algorithms (Support Vector Machine, Decision Tree and Naïve Bayes) and each class implements the abstract class "classifier". In addition, the classes GoodWordAttack, RestrainedAttack, FreeRangeAttack and FeatureDeletionAttack represents adversarial attacks and each class implements the abstract class "attack".

Use Case Diagram

The use case diagram presented in 3.3 consists of a set of possible sequences of interactions between systems and users in the software system. It contains all the system activities that have significance to the users within the software system. In addition, it holds all the functional requirements and represents them in a format that is easy to read and track.

3.4 Development Strategy

Due to the limited time frame in which the software system had to be developed and tested in, a development strategy also called methodology had to be followed to manage the development and testing of the software system and ensure delivering a functioning prototype at the end of the project. Therefore, In this project, Agile software development methodology has been chosen to be the suitable method and was utilised during the implementation and testing phase. Agile development method breaks the project into small incremental builds called iterations. At the end of each iteration, a working product is displayed to the client. Here are some reasons why Agile was chosen over other methods. Firstly, the requirements for this project are at a moderate to high risk of changing. In Agile, any changes to the requirements can be incorporated at any point of the process even in late development processes without the risk of losing the whole work. Secondly, a working software will be delivered much more quickly and successive iterations can be delivered frequently and viewed by the client at a consistent pace. Thirdly, the client will be satisfied by the rapid and continuous delivery of useful software. In addition, feedback will be received from the client after each iteration, which will provide the opportunity to improve the software in the next iterations. Over the course of the project, four iterations has been utilised, each with its own objectives and deliverables and are presented in the following list.

- Iteration One: The first iteration focused on implementing loading, parsing and processing the raw data needed. The deliverable from this iteration was pre-processed dataset that can be used in classification operations.

- Iteration Two: The second iteration focused on creating three classifiers from three different algorithms and allow them to be trained and tested using the dataset resulted from the first iteration. The deliverable from this iteration was the three classifiers being able to be trained and tested.

- Iteration Three: The third iteration focused on creating a range of different attacks that can be used to attack the created classifiers in second iteration. The deliverable from this iteration was four different attacks that can be run during the testing time

of the classifiers.

- Iteration Four: The last iteration focused on the creation of the user interface of the software. the deliverable from this iteration was the user interface that combines the usage of the three previous iterations to provide user interaction with the software.

Chapter 4

Implementation

This section of the report will discuss the implementation of the software system down to the code level. The key parts of the code implemented will be detailed with the related services and tools that were utilised.

4.1 Project Structure

In order to produce a readable, reliable and maintainable software, python modular programming was used to structure the project by splitting the source code into separate small parts [19]. These parts are represented by a python module, which is simply a python file that consists of classes, functions and variable definitions that can be grouped after to produce a complete software system. Hence, the usage of modules facilitates the re-usability of the code and makes it easier to access the code for specific functionality. In addition, similar modules were grouped and categorised into different packages. These packages act as a directory that holds the modules that have similar functionality [19]. The following list demonstrates the final implemented project structure that consists of different packages and modules,

- **DataReader**: This package is responsible for loading, processing, parsing and preparing the raw emails and labels for classification and attacking operations. The main module in this package is *EmailDataset.py*. This module contains a class called **EmailDataset**, which is constructed by loading either raw emails and labels or a serialised object of the class saved in external file (.pkl [1]). If the data is raw, then creating

[1].pkl is a file format of a file that is created by a python module called Pickle, this file contains a serialised object that can be deserialised in run-time

an EmailDataset object will load, process, parse and prepare the data and make it available for access in the software. In addition, all the data saved in the software system are stored in a directory called "Data" that is held in DataReader package.

- **Classifiers**: This package provides an abstract classifier class defined with the necessary operations for using a classifier for email classification problem and three main implementations of the following classifiers, Naïve Bayes, Support Vector Machines and Decision Tree. The list below presents the modules included in classifiers package.
  - *Classifier.py*: Abstract class for the classifiers.
  - *multiNomialNaiveBayesClassifier.py*: classifier representing learning model based on Naïve Bayes classification algorithm.
  - *SupportVectorMachineClassifier.py*: classifier representing learning model based on Support Vector Machines classification algorithm.
  - *DecisionTreeClassifier.py*: classifier representing learning model based on Decision Tree classification algorithm.
- **attacks**: This package provides an abstract class for implementing an attack and is defined in attack.py module and several attack modules that uses the abstract class. Each attack module implemented is presented in the list below and will be discussed in detail in the Attacks section of this chapter.
  - *GoodwordAttack.py*: Contains First N Words and Best N words Attacks.
  - *FreeRangeAttack.py*: Free Range Attack implementation
  - *RestrainedAttack.py*: Restrained Attack implementation
  - *FeatureDeletionAttack.py*: Feature Deletion Attack implementation
- **interface**: This is the main file that uses all the defined modules in data, classifiers and attacks packages to run the software. It is also responsible of creating the command line GUI (Graphical User interface) and specifying the software interactions with the user actions such as, reading user input and presenting results.

4.2 Data Preparation and Processing

4.2.1 Loading the Data

The first step towards the machine learning approaches and methods is loading the necessary data, which consists of raw emails in **.eml** format[2] and their labels in a file with **.label** format provided from **CSDMC2010 SPAM corpus**[3]. Therefore, to load the emails and convert them to strings for further pre-processing and preparing, the function *load_ emails()* has been created as seen in Figure 4.1.

```
#function to load all the emails in a specific directory and convert them to strings
#returns a list of all emails as strings
def load_emails(self, emails_path):
    #list to store the emails
    emails = []
    #loop through each file in the directory
    for file in os.listdir(emails_path):
        email_path = emails_path +"/" +file
        with open(email_path, 'r', encoding="utf8", errors='ignore') as fp:
            email_msg = fp.read()
            emails.append(email_msg)

    return emails
```

Figure 4.1: *load_ emails()* function

This function takes the directory path that holds the raw emails and uses python built-in function *os.listdir()*, which returns a list of the file names in the directory and *open()* to open each file for reading in text mode with the following parameters, *mode* $=$*"r"* for reading, *errors* $=$*"ignore"* to specify that encoding/decoding errors should be ignored and *encoding* $=$*"utf-8"* as it is the most popular encoding for HTML/Text files and the emails might include HTML/Text parts, so it will ensure that each character is decoded correctly. After that, each email file is converted to a string using *read()* function and a list of converted emails is returned. A similar approach has been taken to load the labels of the emails and is implemented in *load_ labels()*.

[2]This file format has the email contents as plain text in MIME format, containing the email header and body

[3]one of the datasets for the data mining competition associated with ICONIP 2010 (International Conference on Neural Information Processing) available from http://csmining.org/index.php/spam-email-datasets-.html. It is composed of a selection of mail messages, suitable for use in testing spam filtering systems.

4.2.2 Email Body Extraction

Next, python **email package**[4] is utilised to parse the emails and extract the body part (also referred to as payload), since it contains the HTML/plain text message, which is the only needed part for our classification problem. Therefore, each email string is converted to a message object structure of the class type **EmailMessage**, which is the base class in email package for an email object model and allows the functionality of accessing email bodies using *email.message_from_string()* function as seen in the implemented function *get_emails_bodies()* in Figure 4.2.

```
#function to extract the body part from the emails
#return a list of emails body part
def get_emails_body(self, emails_path):
    #list to store the emails body
    emails_body = []
    #load the raw emails
    emails = self.load_emails(emails_path)
    for file in tqdm(emails, desc=" Extracting raw email bodies"):
        #convert the email string into a message object structure of class EmailMessage
        email_msg = email.message_from_string(file)
        if email_msg.is_multipart():
            initial_body = ""
            #loop through the payload since the message has multiple payloads
            for payload in email_msg.get_payload():
                body = payload.get_payload()
                #regex to remove HTML tags from the message body
                body = (re.sub("<.*?>", "", str(body))).replace('\n\t', ' ').replace('\n', ' ')
                initial_body += body + ""
            emails_body.append(initial_body)
        else:
            body = email_msg.get_payload()
            body = (re.sub("<.*?>", "", str(body))).replace('\n\t', ' ').replace('\n', ' ')
            emails_body.append(body)

    return emails_body
```

Figure 4.2: *get_emails_bodies()* function

Accessing the email bodies then is achieved by checking first if the email has multiple body parts using *is_multipart()* function. This function returns True if the email has a structured sequence of sub-messages with multiple bodies and False if it is a simple text message. Then, *get_payload()* function is used to return the body of the email (payload), which will be a list that is looped over to create a single body when *is_multipart()* returns True and single

[4]a library for managing email messages available from https://docs.python.org/3/library/email.html

body string otherwise as seen in Figure 4.2. In addition, HTML tags are removed from the bodies using *re.sub()* function from python re module[5] to substitute the pattern `"<.*?>"` with an empty string where `"<"` and `">"` matches HTML tag opening and closing brackets and `".*?"` matches zero or more characters.

4.2.3 Feature Extraction

In all cases of training, testing and attacking the classifiers, operations cannot be performed on the emails body in its raw format. Most Machine Learning algorithms do not expect raw text documents with variable length, but instead numerical fixed size **feature vectors**, where each individual word (token) in an email body occurrence frequency represents a feature and a feature vector of dimension n represents a single email, where n=*number of features* (also called feature space) [16]. Therefore, to solve this problem, **Vectorisation** can be used to turn the raw email bodies into numerical feature vectors using **Bag of words** representation, which performs the following steps,

1. **Tokinising** all the strings and giving numerical ID for each possible token (word) [16].
2. **Counting** the occurrences of the tokens in each document (email) [16].
3. **Normalising** and calculating the weight of the tokens and diminishing importance tokens which appear in the majority of the documents/samples (emails) [16].

Thus, a corpus of emails will be represented by a matrix with one column per token occurring in the corpus and one row per email.

For this purpose, **TfidfVectorizer** class from Scikit-learn has been selected to tokinse, remove stop words that are very present and convert the collection of raw email bodies to a matrix of **Tf-idf** features using Tf-idf term weighting, where **Tf** and **idf** are defined as follows,

- The number of times a term (word) occurs in a given document as Tf.
- Inverse document frequency calculated by $idf(t) = log\frac{1+n_d}{1+d_f(d,t)} + 1$ as idf, where n_d is

[5]This is a python module that provides regular expression matching operations

the number of the documents and $d_f(d,t)$ is the number of the documents that has the term t [16].

Therefore, each feature weight will be calculated as $Tf(t,d) \times idf(t)$. Moreover, Tf-idf term weighting gives higher values to the words that are important and have a significant part in the meaning of the document and minimises of frequently occurring terms such as stop words, which would diminish the frequencies of other terms that are rare but more important to the context if contained in the feature vectors [16]. Therefore, *get_feature_vectors()* function was created as seen in Figure 4.3.

```
#function to transform the raw email bodies into feature vectors
def get_feature_vectors(self, emails_bodies):
    #create a vectoriser
    vectorizer = TfidfVectorizer(analyzer='word', strip_accents=None,
                                 ngram_range=(1, 1), max_df=1.0,
                                 min_df=1, max_features=self.max_features,
                                 binary=False, stop_words='english',
                                 use_idf=True, norm=None)
    #train it on the emails body
    vectorizer = vectorizer.fit(emails_bodies)
    #transform the raw emails body into feature vectors
    features_vectors = vectorizer.transform(tqdm(emails_bodies, desc=" Creating emails feature vector"))
    #created a binarizer that turns the TF-IDF features into binary feature vectors
    # (0 for non occurance and 1 for occurance)
    binarizer = Binarizer().fit(features_vectors)
    #needed for good word attack
    features_bin = binarizer.transform(features_vectors)

    #get the feature names, vocabulary and weights
    feature_names = vectorizer.get_feature_names()
    features_with_indices = vectorizer.vocabulary_
    features_weights = vectorizer.idf_

    return features_vectors, feature_names, features_with_indices, features_weights, features_bin
```

Figure 4.3: *get_feature_vectors()* function

This function starts by creating an object of class TfidfVectorizer with the following parameters, *analyzer='word'* to specify that the features should be made of words, *strip_accents=None* to specifiy that no accents need to be removed during pre-processing, *ngram_range=(1, 1)*, *max_features=self.max_features* to specify the max number of features to be extracted, *stop_words='english'* to remove english stop words and *norm=None* so no normalisation is applied. Then, the created vectoriser is trained on the raw email bodies to learn the vocabulary and idf using *fit()* and turn the raw email bodies into document-term matrix using *transform()* function. The returned document-term matrix is represented by a compressed sparse row matrix data structure, where each row is a feature vector for a single email and

each column is a word. The reason why compressed sparse row matrix is used is because it does not require so much space in the memory and algebraic operations can be performed faster on it. An additional feature that can be seen in Figure 4.3 is using Scikit-learn **Binarizer** class. This class has been used to transform the feature vectors into binary feature vectors, where the occurrence of a word in a document is set to 1 if its present and to 0 otherwise. This has been added as some implemented attacks require having binary feature vectors.

4.2.4 Splitting the Dataset

After vectorising the raw emails, the feature vectors are split into two separate subsets, **training** set and **testing** set with having 70 % for training and 30 % for testing. The training set will be used to train the models alongside with their labels and the testing set will be used to test the predictions of the models. This has been done to avoid Overfitting the models as discussed in the Chapter 2. Thus, *split_ data()* function has been implemented to handle splitting the data and is presented in Figure 4.4.

```
#function to split the features and labels into training and testing subsets
def split_data(self, features, labels, test_size =0.3):

    x_train, x_test, y_train, y_test = train_test_split(features, labels, test_size=test_size, random_state=42)

    return  x_train, x_test, y_train, y_test
```

Figure 4.4: *split_ data()* function

This function uses the helper function *train_ test_ split()* provided from Scikit-learn library, which takes the feature vectors and their labels and returns two subsets with two lists that holds their labels after shuffling them and splitting them according to the parameter *test_ size*, which specifies the percentage sizes of the subsets as seen in Figure 4.4.

4.2.5 Saving and Loading EmailDataset

Pickle python library is used to save (serialise) and load (deserialise) previously created EmailDataset object as it provides simple and fast performance. For saving the EmailDataset object, pickle converts the object into a byte stream in order to store it in an external

file using *pickle.dump()* function. For loading, it reads the byte stream and recreates the original object hierarchy and populates it with the data using instructions included in the byte stream that aids in recreating the same object using *pickle.load()*.

4.3 Classifiers

Three types of classifiers have been created for this software, these classifiers have been constructed as classes that implement the abstract class **classifier** provided in *Classifier.py* and each classifier implements a suitable class of its algorithm learning model from Scikit-learn classifiers. Therefore, for Naïve Bayes classifier, Scikit-learn **MultinomialNB** has been chosen to represent the model, for Support Vector Machines classifier, **SVC** has been chosen to represent the model and finally for Decision Tree classifier, **DecisionTreeClassifier** has been chosen.

4.3.1 Training and Testing the Classifiers

For training and testing the classifiers, *fit()* and *predict()* functions has been used and are provided by Scikit-learn. The function *fit()* is used on a classifier to train it and fit it to the training set. Therefore, it requires passing the training set instances and the labels. After fitting the model, the function *predict()* is used on the classifier to classify incoming data points from a testing set or other previously unseen data. Figure 4.5 shows how these functions are used.

```
classifiers = [multiNomialNaiveBayesClassifier(), decisionTreeClassifier(), svmClassifier()]
names = ["Naive Bayes Classifier:", "Decision Tree Classifier:", "Support Vector Machines Classifier:"]
count = 0
for classifier in classifiers:
    clf = classifier
    clf.fit(dataset.x_train, dataset.y_train)
    pred = clf.predict(dataset.x_test)
    print(Fore.LIGHTCYAN_EX + names[count] + "\n")
```

Figure 4.5: *fit() and predict()* functions

4.3.2 Evaluating the Classifiers

For evaluating the classifiers as discussed in the background chapter, Scikit-learn Confusion Matrix, Accuracy, Precision, and Recall classification metric functions has been used. Each of these functions require passing the true (correct) labels with the corresponding predicted labels. The list below shows the evaluation metric with the function used from Scikit-learn and Figure 4.6 shows how they called in the code,

- Confusion Matrix: *confusion_ matrix()*
- Accuracy: *accuracy_ score()*
- Precision: *precision_ score()*
- Recall: *recall_ score()*

```
print(Fore.LIGHTCYAN_EX + " (*) " + Fore.RESET + "Confusion Matrix: \n " + Fore.LIGHTGREEN_EX + str(
    confusion_matrix(dataset.y_test, ad_preds, labels=[-1, 1])))
print(Fore.LIGHTCYAN_EX + " (*) " + Fore.RESET + "Accuracy: " + Fore.LIGHTGREEN_EX + str(
    accuracy_score(dataset.y_test, ad_preds)))
print(Fore.LIGHTCYAN_EX + " (*) " + Fore.RESET + "Precision: " + Fore.LIGHTGREEN_EX + str(
    precision_score(dataset.y_test, ad_preds, average='binary')))
print(Fore.LIGHTCYAN_EX + " (*) " + Fore.RESET + "Recall: " + Fore.LIGHTGREEN_EX + str(
    recall_score(dataset.y_test, ad_preds, average='binary')))
```

Figure 4.6: *Evaluation Metrics code*

4.4 Attacks

4.4.1 Free Range Attack & Restrained Attack [21]

Free Range and Restrained attacks are implemented using the Adversarial Support Vector Machine Learning proposed algorithms by Yan Zhou, Murat Kantarcioglu, Bhavani Thuraisingham and Bowei Xi and Adversarial Machine Learning Library (Ad-lib). These attacks goal is to move the spam instances features by a specific distance, this distance is measured as the harshness of the attack. In Free Range attack, features can be moved anywhere in the feature space, while it is more conservative in Restrained attack [21].

Free Range Attack [21]

Free Range attack aims to move the feature values in the spam instances anywhere in the feature space. It requires finding a valid range of the minimum and maximum values of each feature in the feature space of all the testing set instances [21]. Therefore, this has been implemented in *min_feature_val()* and *max_feature_val()* functions. These functions simply iterate over each feature in the feature space for each single instance and tries to find the maximum and minimum value a feature could have in the testing set instances. After finding the valid ranges of maximum and minimum values of each feature, the following formula is applied in the function *transform_feature_vector()* (Figure 4.7) to find δ_{ij}, which is the displacement value for a data point x_{ij},

$$C_f(x_{ij}^{min} - x_{ij}) \leq \delta_{ij} \leq C_f(x_{ij}^{max} - x_{ij}), \forall j \in [1, d],$$

where $C_f \in [0, 1]$ controls the aggressiveness of attacks. $C_f = 0$ means no attacks, while $C_f = 1$ corresponds to the most aggressive attacks involving the widest range of permitted data movement and x_{ij}^{min} and x_{ij}^{max} are the smallest and the largest values that the jth feature of an instance (xi) can take [21]. After finding the displacement value δ_{ij}, it is added to the original value of x_{ij} as seen in Figure 4.7.

```
#function to transform feature vectors into adversarial examples
def transform_feature_vector(self, feature_vector, min_values, max_values):

    feature_vector_copy = deepcopy(feature_vector)
    for index in range(0, self.features_num):
        #get the jth feature value
        xij =  feature_vector_copy[index]
        #set the lower and upper bounds of the jth feature value
        lower_bound = self.attack_power * (min_values[index] - xij)
        upper_bound = self.attack_power * (max_values[index] - xij)
        #find delta_ij
        delta_ij = random.uniform(lower_bound, upper_bound)
        #add delta_ij value to the original value of the jth feature
        feature_vector_copy[index] = xij + delta_ij


    return feature_vector_copy
```

Figure 4.7: *transform_feature_vector()* function

Therefore, as seen in Figure 4.7, the function starts by looping through each possible feature in the feature space and sets its original value in the variable *xij* (x_{ij} in the formula). Then, it finds the lower and upper bounds of the feature by multiplying the attribute *attack_power* (C_f in the formula and has default value= 1) by the minimum value the feature has and the maximum value. After that, the displacement value δ_{ij} presented by the variable *delta_ij* is found using python built-in function *random.uniform()* with the range (*lower_bound, upper_bound*), which returns a random value between the specified range. This value is then added to the variable *xij* and is set in the feature vector. This is repeated for each feature in each spam instance using the *attack()* function.

Restrained Attack [21]

Restrained attack aims to move the feature values in the spam instances close to a chosen target ham instance. Therefore, to initiate this attack, a ham instance must be set as a target instance that the feature values will be pushed to [21]. According to Zhou, this can be chosen randomly or can be an estimate centroid of innocuous data (ham instances), a point sampled from the observed innocuous data, or an artificial data point generated from the estimated innocuous data distribution [21]. Thus, the function *find_centroid()* presented in Figure 4.8 creates an artificial ham instance by constructing a Compressed Sparse Row matrix data structure with dimension n = *feature space*. Therefore, *csr_matrix* data structure provided by **scipy** python package has been used to create the artificial ham instance. *csr_matrix* requires three parameters to be created, each parameter is provided in the list below with an explanation of how it was obtained,

- data: This is a list that contains the value of each feature in the feature space and for our artificial ham instance, each feature value equals the average of its feature values in all ham instances in the testing set. Therefore, the variable *sum* has been created and it is incremented in each ham instance by the value of the feature in a specific index. The sum is then divided for each feature by the feature space to get the average and is appended to the list *data*.
- indices: This list contains the indices of the features in the same order as they appear

in the *data* list. Therefore, while looping through the feature space, each index has been appended to this list.

- indptr: This specifies where the row starts in the *data* and in the *indices* lists and is implemented as a pointer array with the range (0 and the length of *indices*.

```
def find_centroid(self, train_set, train_labels):
    self.features_num = len(train_set[0].toarray()[0])
    indices = []
    data = []
    for index in range(0, self.features_num):
        sum = 0
        count = 0
        for feature_vector in train_set[:20]:
            if train_labels[count] == -1:
                sum += feature_vector.toarray()[0][index]
                count +=1
            else:
                count +=1
        sum /= self.features_num
        if sum != 0:
            indices.append(index)
            data.append(sum)
    indptr = [0, len(indices)]
    centroid = csr_matrix((data, indices, indptr), shape=(1, self.features_num))
    return centroid
```

Figure 4.8: *find_ centroid()* function

This artificial ham instance is then classified by the classifier in *set_ innocuous_ target()* function, if the result is spam then this artificial instance is not used and a random ham instance is selected from the training set and set to be the target instance. Next, each spam instance is transformed by changing each feature value by adding a displacement value to it specified in the variable *delta_ ij* (δ_{ij}). This displacement value δ_{ij} is any value between the following range,

$$0 \leq \delta_{ij} \leq C_{\xi}\left(1 - C_{\delta}\frac{\mid x_{ij}^{t} - x_{ij} \mid}{\mid x_{ij} \mid + \mid x_{ij}^{t} \mid}\right),$$

where $C_{\xi} \in [0, 1]$ is a constant that represents the data movement factor ($C_{\xi} = 1$ leads to high amount of data movement while $C_{\xi} = 0$ leads to narrower limit on data movement), $C_{\delta} \in [0, 1]$ is a constant that models the loss of malicious utility as a result of the movement (δ_{ij}) ($C_{\delta} = 1$ leads to less aggressive attacks while $C_{\delta} = 0$ leads to more aggressive attacks) and x_{ij}^{t} is the

jth feature value in the artificial ham instance [21]. Therefore, in *transform_feature_vector()* function (Figure 4.9), a random value is selected between 0 and the variable *bound*, which is calculated as $C_\xi\left(1 - C_\delta \frac{|x_{ij}^t - x_{ij}|}{|x_{ij}| + |x_{ij}^t|}\right)$ by having $data\_movement\_factor = 1$ (C_ξ) and $attack\_power = 0$ (C_δ) for each feature and is added to the original value variable *xij* as in the Free Range Attack and again it is repeated for each spam instance using the *attack()* function to create the adversarial set.

```
def transform_feature_vector(self, feature_vector):

    feature_vector_copy = deepcopy(feature_vector)
    for index in range(0, self.features_num):
        #if index in transform_features_index:
            xij =  feature_vector_copy[index]
            target = self.innocuous_target.toarray()[0][index]
            if abs(xij) + abs(target) == 0:
                bound = 0
            else:
                bound = self.data_movement_factor * (1 - self.attack_power *(abs(target - xij)
                        /(abs(xij) + abs(target)))) * abs((target - xij))
            delta_ij = random.uniform(0, bound)
            feature_vector_copy[index] = xij + delta_ij

    return feature_vector_copy
```

Figure 4.9: *transform_feature_vector()* function

4.4.2 Feature Deletion Attack [3]

Based on Nightmare at Test Time: Robust Learning by Feature Deletion by Amir Globerson and Sam Roweis and Adversarial Machine Learning Library (Ad-lib), this attack aims to find a list of features of length n, which might be the most harmful to a certain given classifier and sets their value in the spam instances to 0, which denotes a deleted feature [3]. These features could be the features with least weights or highest weights assigned by the classifier or even randomly selected [3]. In this implementation, features with the least weights are selected and deleted from the spam instances. Since we have different classifiers, accessing the weights assigned to the features differ and had to be taken in consideration of the implementation. Therefore, this attack implementation starts with calling the function *get_del_features_index()* (Figure 4.10), which checks the name of the classifier if its either MultinomialNB , SVM or DT for Decision tree and then uses the correct attribute that gets the weights assigned to the feature by that classifier, which is *coef_* attribute for both

Naive Bayes and SVM but differ in their internal construction and *feature_importances_* for Decision tree as seen in Figure 4.10.

```
def get_del_features_index(self, trained_classifier, del_num):

    if self.name == "MultinomialNB":
        del_features_index = np.argsort(trained_classifier.coef_[0])[:del_num]
        return del_features_index
    elif self.name == "SVM":
        del_features_index = np.argsort(trained_classifier.coef_[0].toarray()[0])[:del_num]
        return del_features_index
    elif self.name == "DT":
        del_features_index = np.argsort(trained_classifier.feature_importances_)[:del_num]

        return del_features_index
```

Figure 4.10: *get_del_features_indices()* function

The features are then sorted in ascending order and the indices of these features are returned using *np.argsort()* and then using python slicing method *[:del_num]* a list with length *del_num* (Number of features to be deleted has default value = 1000) of indices will be returned. The next step is to set the values of the selected features to 0 in the testing spam instances. Therefore, the function *del_features()* (Figure 4.11) takes the list of the feature indices to be deleted and a single spam instance feature vector and loops through all possible features and check if the feature index is in the list of the feature indices to be deleted and sets its value to 0 if its included and then returns the new copy of the feature vector. This step is then repeated for each spam instance feature vector and an adversarial set is created with all the forged spam instances and the ham instances using *attack()* function.

```
def del_features(self, feature_vector, del_features_index):

    feature_vector_copy = deepcopy(feature_vector)
    for index in range(0, self.features_num):

        if index in del_features_index:
            feature_vector_copy[index] = 0

    return  feature_vector_copy
```

Figure 4.11: *del_features()* function

4.4.3 Good Word Attacks [13]

This attack implementation is adopted from Good Word Attacks on Statistical Spam Filters by Daniel Lowd and Christopher Meek [13]. The general idea behind this attack is to modify spam emails by appending words that mostly appear in ham emails [13]. Therefore, the attack requires finding a list of words that are considered to be strongly ham to a certain classifier. Two types of the attack have been implemented, **First-N Words** and **Best-N Words**. Both types of the attacks require finding a pair of spam/ham emails that differ by one word only [13]. This is performed by eliminating words that are not present in the spam email words from the ham email and then adding the ones from spam email words to it until it is classified as spam. For this purpose, the algorithm in Figure 4.12 was adopted in the implementation of *find_ witness()* function.

$M_{curr} \leftarrow M_{legit}$
repeat
  $M_{prev} \leftarrow M_{curr}$
  if some word w is in M_{curr} but not M_{spam} **then**
    remove w from M_{curr}
  else if some w is in M_{spam} but not M_{curr} **then**
    add w to M_{curr}
  end if
until M_{curr} classified as spam
return (M_{curr}, M_{prev})

Figure 4.12: Find witness algorithm, originally from [13]

In detail, *find_ witness()* function starts by setting the values of *current_ message*, which represents M_curr in the algorithm as the ham email feature vector and the *spam_ message*, which represents M_spam as the spam email feature vector. Then, it extracts the word indices from *current_ message* and *spam_ message* using *get_ spam_ ham_ words()* function (Figure 4.13) and returns them as two sets, one represents the ham words *curr_ words* and the other represents the spam words *spam_ words*. Next, a while loop is initiated. This while loop checks if *curr_ message* is classified as spam by the classifier that is attacked each time after adding a word from *spam_ words* set and after removing all the words from *curr_ words* that do not appear in *spam_ words*. In each iteration in the loop, *prev_ message* is set to a

copy of *current_message*, this is because if a spam word was added to *current_message* and it was classified as spam, then we want to know the message which differs from it by one word, which is defined in *prev_message*. After finding *curr_message* that is classified as spam, the loop terminates and the pair of *curr_message* and *prev_message* are returned. After implementing *find_witness()*, Both First_N words and Best_N words algorithms have been implemented in *first_n_words()* and *best_n_words()* functions.

```
def get_spam_ham_words(self, spam_email, ham_email):

    spam_email_words = set()
    ham_email_words = set()
    for index in range(0,self.features_num):
        if int(spam_email.toarray()[0][index]) != 0:
            spam_email_words.add(index)

    for index in range(0,self.features_num):
        if int(ham_email.toarray()[0][index]) != 0:
            ham_email_words.add(index)


    return spam_email_words, ham_email_words
```

Figure 4.13: *get_spam_ham_words()* function

The first algorithm First-N words (Figure 4.14) finds a fixed length list of good words by testing the classification of the spam message after the addition of each single word from the word space. If the classification of the message is ham then the word is considered as a good word. This is implemented in function *first_n_words()*, which starts by creating an empty set to hold the good words that will be found and by calling *find_witness()* to set the spam and ham messages that differ by one word. Then, it loops through each feature in our feature space and checks if the word is present in the spam message. If the word is not present, then it is added to the spam message and the spam message is then classified using the classifier. If the classification of the spam message after the addition of the word is ham, then the word is added to the list of good words. This is repeated until all the words are tested or until the length of the good words list equals the attribute *num_good_words* which is set to 500 in default. After that, a list of good word is returned and is used in the *attack()* function. Finally, the *attack()* function is called to create an adversarial set by using all the

good words found by First_N words algorithm and appending them to the spam instances in the testing set. As in the First-N Words attack, the Best N Words (Figure 4.15) attack

$L \leftarrow \emptyset$
$(M_{\epsilon+}, M_{\epsilon-}) = \text{FINDWITNESS}(M_{spam}, M_{legit})$
for each word $w \in W$ **do**
  if $M_{\epsilon+}$ with w classified as legitimate **then**
    add word w to L
  end if
  if L contains at least n words **then**
    exit loop
  end if
end for
return L

Figure 4.14: First-N Words algorithm, originally from [13]

builds a list of ham words by adding each feature in the feature space to the ham message and then classify it by the classifier. However, it also builds a list of spam words. The list of the spam words is constructed from adding each word in our feature space to the ham email and then classifying it by the classifier. If it is classified as spam, then the word is added to the list of spam words. Each word in the spam words then partitions the list of ham words into two sets, one with the ham words that have more importance than the spam word , and one with the ham words that have less importance than the spam word. This is done by adding each single spam word in the ham message and adding each word from the ham words in each turn for that spam word, then it is classified by the classifier, if the result is spam , then it means that the added ham word has more importance than the added spam word and its added to the list with small weights, otherwise, if it is classified as ham then the ham word has less effect or importance and is added to the list of large weights. This is done for each spam word in spam words each turn with testing all the ham words. Since the goal is to find the best n words, the algorithm then reduces the set of negative words under consideration. If the set of greater magnitude words is larger than n, then it never needs to consider any of the lesser magnitude words. On the other hand, if the set of greater magnitude words is smaller than n, then this entire set is a subset of the best n words, so the algorithm focuses future iterations on the less-negative set.The algorithm halts when it has found the n best words, or when 10 positive words in a row yield no progress. In the latter case, its list of n words is a combination of the best words found and random negative words still under consideration.

$(M_{e+}, M_{e-}) = \text{FINDWITNESS}(M_{spam}, M_{legit})$
$S \leftarrow \{w \in W | M_{e-}$ with w is classified as spam$\}$
$L \leftarrow \{w \in W | M_{e+}$ with w is classified legitimate$\}$
$L_{best} \leftarrow \emptyset$
for each spammy word $w_+ \in S$ **do**
  $L_{small} \leftarrow$
  $\{w_- \in L | M_{e+}$ with w_+, w_- classified as spam$\}$
  $L_{large} \leftarrow$
  $\{w_- \in L | M_{e+}$ with w_+, w_- classified legitimate$\}$
  if $|L_{best}| + |L_{large}| < n$ **then**
    remove all words in L_{large} from L
    add all words in L_{large} to L_{best}
  else
    remove all words in L_{small} from L
  end if
  if L remains unchanged for 10 iterations **then**
    augment L_{best} with $n - |L_{best}|$ words from L
    exit loop
  end if
end for
return L_{best}

Figure 4.15: Best-N Words algorithm, originally from [13]

4.5 Interface

The interface of the software as discussed in the specification and design chapter has been implemented as command line GUI. Therefore, the interface consists of printed text using python *print()* function, and taking user input using python *input()* function to interact with the users and allow the ability of choosing different functions. To make the experience more interesting and attractive, two packages has been used as seen below,

- Colorama: This package allows using colours for the printed text on the command line interface, this is helpful because it allows the user to differentiate between a command and another. This package has been used by importing two functions, *init* and *Fore*. *init()* is used to initiate using the colours and Fore is used to set the foreground colour of the text. Figure 4.16 shows how the instructions of the software are printed using

the colour light green and red. Figure 4.17 shows a screen from the software system that utilises Colorama.

- Tqdm: This package has been used to show a progress bar (loading) in the interface when a function is using a loop that might take long time. Using it is simple, it is wrapped around and iterable that is in a while or for loop. This approach is good, as some functions take long time, and we want the user to know what is going on.

```
def instructions():
    print("\n")
    print(Fore.LIGHTGREEN_EX + " Welcome to Adversarial Reasoning in Machine Learning for Natural "
                               "Language Processing: The Case of Spam Emails!")
    print(Fore.LIGHTRED_EX + "\n *Important*" + Fore.RESET + " This software requires specifying the path"
          + " of the folder which holds the Spam and Ham emails and the" + "\n path of the file which holds "
            "the labels of those emails for creating preprocessed dataset for training and \n testing "
            "classifiers. If you already have a saved dataset, please select no when asked for creating "
             "new dataset\n and specify the path to the dataset.")
```

Figure 4.16: Usage of Colorama

```
##########################################################################################################################
#        Adversarial Reasoning in Machine Learning for Natural Language Processing: The Case of Spam Emails       #
##########################################################################################################################
--------------------------------------------------------------------------------------------------------------------------
                                              Evaluation summary on all classifiers
--------------------------------------------------------------------------------------------------------------------------
                          Dataset: 3028 Training emails - 1299 Testing emails - 4327 emails in total
--------------------------------------------------------------------------------------------------------------------------
Naive Bayes Classifier:

 (*) Confusion Matrix:
[[858  13]
 [ 54 374]]
 (*) Accuracy: 0.948421862972
 (*) Precision: 0.966408268734
 (*) Recall: 0.873831775701


Decision Tree Classifier:

 (*) Confusion Matrix:
[[835  36]
 [ 31 397]]
 (*) Accuracy: 0.948421862972
 (*) Precision: 0.916859122402
 (*) Recall: 0.927570093458


Support Vector Machines Classifier:

 (*) Confusion Matrix:
[[854  17]
 [ 30 398]]
 (*) Accuracy: 0.963818321786
 (*) Precision: 0.959036144578
 (*) Recall: 0.929906542056



 - Would you like to go back to main menu? (Y/N)
```

Figure 4.17: Usage of Colorama 2

Chapter 5

Software System Testing

To demonstrate that the software system developed works as intended and meets the system requirements, test cases have been created and carried out on the most crucial functions in the software system. Each test case has been carried out on a Windows 10 operating system device using Windows Command Prompt to run the software and is presented below.

Test Cases

Test Case Id: TC-1	**Test Purpose:** Loading raw emails, labels and creating a pre-processed dataset		
Preconditions: None			
Test Case Steps: 3			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "Y" when prompted with creating a new dataset from raw emails and labels	A message should appear to prompt the user to input the path of raw emails	Pass
2	Input the path of the raw emails directory when prompted and click enter	Another message should appear asking the user to input the email labels file path	Pass
3	Input the path of the labels file when prompted and click enter	A progress bar should show up presenting loading the files and a message showing successful creation of the dataset	Pass
Comments: This test case passed all steps			
Related Tests: -			

Table 5.1: TC-1 Loading raw emails, labels and creating a pre-proccessed dataset

Test Case Id: TC-2	**Test Purpose:** Loading an existing pre-processed dataset		
Preconditions: An existing dataset that was previously saved by running TC-3 should be available to load			
Test Case Steps: 2			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "N" when prompted with creating a new dataset from raw emails and labels and click enter	A message should appear to prompt the user to input the path of the saved dataset	Pass
2	Input the path of dataset (e.g. dataset.pkl) and click enter	A message should appear specifying that the dataset is being loaded	Pass
		A new screen is showed with the main menu and the dataset details on the top banner	Pass
Comments: This test case passed all steps			
Related Tests: TC-3			

Table 5.2: TC-2 Loading an existing pre-processed dataset

Test Case Id: TC-3	**Test Purpose:** Saving a dataset		
Preconditions: Created a dataset from raw emails and labels successfully by running TC-1			
Test Case Steps: 2			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "Y" when prompted to save the dataset and click enter	A message should appear to prompt the user to input a name for the file	Pass
2	Input the desired name for the dataset that will be saved (e.g. dataset2) and click enter	A message should appear specifying that the dataset has been successfully created	Pass
Comments: This test case passed all steps			
Related Tests: TC-1			

Table 5.3: TC-3 Saving a dataset

Test Case Id: TC-4	**Test Purpose:** Creating, training and testing all classifiers (Showing a summary of the testing results from all classifiers)		
Preconditions: Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2			
Test Case Steps: 1			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "2" when prompted to choose the option "Create and test all classifiers" and click enter	A new screen should show up with the testing evaluation summary for all the classifiers	Pass
Comments: This test case passed all steps			
Related Tests: TC-1 and TC-2			

Table 5.4: TC-4 Creating, training and testing all classifiers (Showing a summary of the testing results from all classifiers)

Test Case Id: TC-5	**Test Purpose:** Creating, training and testing all classifiers in presence of each possible attack (Showing a summary of the testing results from all classifiers after each possible attack)		
Preconditions: Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "3" when prompted to choose the option "Create and attack all classifier with all possible attacks" and click enter	A new screen should show up with the testing evaluation summary for all the classifiers after each possible attack	Pass
Comments: This test case passed all steps			
Related Tests: TC-1 and TC-2			

Table 5.5: TC-5 Creating, training and testing all classifiers in presence of each possible attack (Showing a summary of the testing results from all classifiers after each possible attack)

Test Case Id: TC-6	**Test Purpose:** Creating and training Naïve Bayes classifier		
Preconditions: Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "1" when prompted to choose the option "Create a new Classifier" and click enter	A new screen should show up with list of possible classifiers to create	Pass
2	Input "1" when prompted to choose the option "Naive Bayes Classifier" and click enter	A new screen should show up with a banner having "Naive Bayes" as the title and a menu with several options	Pass
Comments: This test case passed all steps			
Related Tests: TC-1 and TC-2			

Table 5.6: TC-6 Creating and training Naïve Bayes classifier

Test Case Id: TC-7	**Test Purpose:** Creating and training Decision Tree classifier		
Preconditions: Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "1" when prompted to choose the option "Create a new Classifier" and click enter	A new screen should show up with list of possible classifiers to create	Pass
2	Input "2" when prompted to choose the option "Decision Tree Classifier" and click enter	A new screen should show up with a banner having "DT" as the title and a menu with several options	Pass
Comments: This test case passed all steps			
Related Tests: TC-1 and TC-2			

Table 5.7: TC-7 Creating and training Decision Tree classifier

Test Case Id: TC-8	**Test Purpose:** Creating and training Support Vector Machine classifier		
Preconditions: Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "1" when prompted to choose the option "Create a new Classifier" and click enter	A new screen should show up with list of possible classifiers to create	Pass
2	Input "3" when prompted to choose the option "Support Vector Machines Classifier" and click enter	A new screen should show up with a banner having "SVM" as the title and a menu with several options	Pass
Comments: This test case passed all steps			
Related Tests: TC-1 and TC-2			

Table 5.8: TC-8 Creating and training Support Vector Machine classifier

Test Case Id: TC-9	**Test Purpose:** Creating and training a Black Box classifier		
Preconditions: Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "1" when prompted to choose the option "Create a new Classifier" and click enter	A new screen should show up with list of possible classifiers to create	Pass
2	Input "4" when prompted to choose the option "Black Box Classifier" and click enter	A new screen should show up with a banner having "Black Box Classifier" as the title and a menu with several options	Pass
Comments: This test case passed all steps			
Related Tests: TC-1 and TC-2			

Table 5.9: TC-9 Creating and training a Black Box classifier

Test Case Id: TC-10	**Test Purpose:** Saving a trained classifier		
Preconditions: 1- Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2 2- Created a classifier by running one of the following tests, TC-6, TC-7 or TC-8			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "4" when prompted to choose the option "Save Classifier" and click enter	A new message will show up asking to input the name of the file to be saved to	Pass
2	Input a name for the file and click enter	A message will show up indicating that the file has been saved successfully and the current screen will reload	Pass
Comments: This test case passed all steps			
Related Tests: TC-1, TC-2, TC-6, TC-7 and TC-8			

Table 5.10: TC-10 Saving a trained classifier

Test Case Id: TC-11	**Test Purpose:** Test a trained Classifier		
Preconditions: 1- Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2 2- Created a classifier by running one of the following tests, TC-6, TC-7 or TC-8			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "1" when prompted to choose the option "Test Classifier" and click enter	A new screen should show up with the testing evaluation summary for the chosen classifier	Pass
Comments: This test case passed all steps			
Related Tests: TC-1, TC-2, TC-6, TC-7 and TC-8			

Table 5.11: TC-11 Test a trained Classifier

Test Case Id: TC-12	**Test Purpose:** Predict the labels of unseen emails using a trained classifier		
Preconditions: 1- Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2 2- Created a classifier by running one of the following tests, TC-6, TC-7 or TC-8			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "2" when prompted to choose the option "Predict the label of Unseen Email" and click enter	A new screen should show up with a message asking to input the path directory, which holds the emails	Pass
2	Input the directory path and click enter	A list of email file names alongside with the predicted labels will be presented	Pass
Comments: This test case passed all steps			
Related Tests: TC-1, TC-2, TC-6, TC-7 and TC-8			

Table 5.12: TC-12 Predict the labels of unseen emails using a trained classifier

Test Case Id: TC-13	**Test Purpose:** Load an existing classifier		
Preconditions: 1- Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2 2- Having an existing classifier that was saved by running TC-10			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "1" when prompted to choose the option "Create a new Classifier" and click enter	A new screen should show up with list of possible classifiers to create	Pass
2	Input "5" when prompted to choose the option "Load Existing Classifier" and click enter	A new message will show up asking to input the name of the file of the existing classifier	Pass
3	Input the name of the file and click enter	A message will show up indicating that the classifier has been successfully loaded and then a new screen will show up with the loaded classifier details with a menu of several options	Pass
Comments: This test case passed all steps			
Related Tests: TC-1, TC-2 and TC-10			

Table 5.13: TC-13 Load an existing classifier

Test Case Id: TC-14	**Test Purpose:** Attack a classifier using Good Word Attack		
Preconditions: 1- Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2 2- Created a classifier by running TC-6, TC-7 or TC-8			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "3" when prompted to choose the option "Attack Classifier" and click enter	A new screen should show up a menu with possible attacks	Pass
2	Input "1" when prompted to choose the option "Good word Attack" and click enter	A new screen will show up with the results of testing the selected classifier in presence of Good Word attack	Pass
Comments: This test case passed all steps			
Related Tests: TC-1, TC-2, TC-6, TC-7 and TC-8			

Table 5.14: TC-14 Attack a classifier using Good Word Attack

Test Case Id: TC-15	**Test Purpose:** Attack a classifier using Feature Deletion Attack		
Preconditions: 1- Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2 2- Created a classifier by running TC-6, TC-7 or TC-8			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "3" when prompted to choose the option "Attack Classifier" and click enter	A new screen should show up a menu with possible attacks	Pass
2	Input "4" when prompted to choose the option "Feature deletion Attack" and click enter	A new screen will show up with the results of testing the selected classifier in presence of Feature Deletion attack	Pass
Comments: This test case passed all steps			
Related Tests: TC-1, TC-2, TC-6, TC-7 and TC-8			

Table 5.15: TC-15 Attack a classifier using Feature Deletion Attack

Test Case Id: TC-16	**Test Purpose:** Attack a classifier using Free Range Attack		
Preconditions: 1- Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2 2- Created a classifier by running TC-6, TC-7 or TC-8			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "3" when prompted to choose the option "Attack Classifier" and click enter	A new screen should show up a menu with possible attacks	Pass
2	Input "2" when prompted to choose the option "Free range Attack" and click enter	A new screen will show up with the results of testing the selected classifier in presence of Free Range attack	Pass
Comments: This test case passed all steps			
Related Tests: TC-1, TC-2, TC-6, TC-7 and TC-8			

Table 5.16: TC-16 Attack a classifier using Free Range Attack

Test Case Id: TC-17	**Test Purpose:** Attack a classifier using Restrained Attack		
Preconditions: 1- Created a dataset from raw emails and labels successfully by running TC-1 or loaded an existing dataset by running TC-2 2- Created a classifier by running TC-6, TC-7 or TC-8			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "3" when prompted to choose the option "Attack Classifier" and click enter	A new screen should show up a menu with possible attacks	Pass
2	Input "3" when prompted to choose the option "Restrained Attack" and click enter	A new screen will show up with the results of testing the selected classifier in presence of Restrained attack	Pass
Comments: This test case passed all steps			
Related Tests: TC-1, TC-2, TC-6, TC-7 and TC-8			

Table 5.17: TC-17 Attack a classifier using Restrained Attack

Test Case Id: TC-18	**Test Purpose:** Save an adversarial set after attacking		
Preconditions: Attack by running any of the following tests, TC-14, TC-15, TC-16 or TC-17			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "Y" when prompted to save the resulting adversarial set from a chosen attack on a specific classifier and click enter	A message should show up asking to input the name of the file to be saved	Pass
2	Input the name of the file that will be saved and click enter	A message will appear stating that the set has been successfully saved	Pass
Comments: This test case passed all steps			
Related Tests: TC-14, TC-15, TC-16 and TC-17			

Table 5.18: TC-18 Save an adversarial set after attacking

Test Case Id: TC-19	**Test Purpose:** Load and attack using existing adversarial set		
Preconditions: 1- Having an existing adversarial set saved by running TC-18 2- Created and trained a classifier by running TC-6, TC-7, TC-8 or TC-9 or loaded an existing classifier by running TC-13			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Input "3" when prompted to choose the option "Attack Classifier" and click enter	A new screen should show up with a menu of possible attacks	Pass
2	Input "5" when prompted to choose the option "Load saved attack" and click enter	A new screen will show up with a message asking to input the name of the file where the attack is saved	Pass
3	Input the name of the file of an existing attack and click enter	The results from testing the classifier with the attack will be showed	Pass
Comments: This test case passed all steps			
Related Tests: TC-18, TC-13, TC-9, TC-6, TC-7 and TC-8			

Table 5.19: TC-19 Load and attack using existing adversarial set

It can be seen from the test cases, that all of them passed with no errors present during the testing, which means that all the requirements were satisfied and met. Therefore, it indicates that the approaches taken to solve the main problem of the project and implement the software system have been a good choice and resulted in delivering a successful software system.

Chapter 6

Results

After testing the implementation of the software system, and founding out that all tests have passed and the system requirements were met successfully, it is possible now to evaluate the classifiers using the software system to analyse their performance in presence of an adversary.

6.1 Classifiers Performance before Attacks

First, it is important to the identify performances of each classifier when acting as a spam filter having the dataset from CSDMC2010 SPAM corpus. This is necessary because it will indicate the classifier that works well on this dataset as a spam filter. In addition, the results can be used to compare the performances of the classifiers before and after the presence of an adversary. Therefore, each classifier has been created, trained and tested multiple times using different random states of training and testing sets. the evaluation metrics that were focused on are Accuracy and Recall and are discussed below.

6.1.1 Accuracy

Figure 6.1 has been created with the accuracy scores that were obtained from testing the classifiers in six different tries. It can be clearly seen that Support Vector Machine Classifier (SVM) outperforms both Naïve Bayes (NB) and Decision Tree Classifier (DT) in regards to the accuracy scores. Unlike DT and NB who maintain an accuracy score between 0.939 and 0.956, SVM maintains an accuracy score between 0.956 and 0.968, which is considered as the highest among the classifiers. In addition, it can be noticed that all the classifiers perform

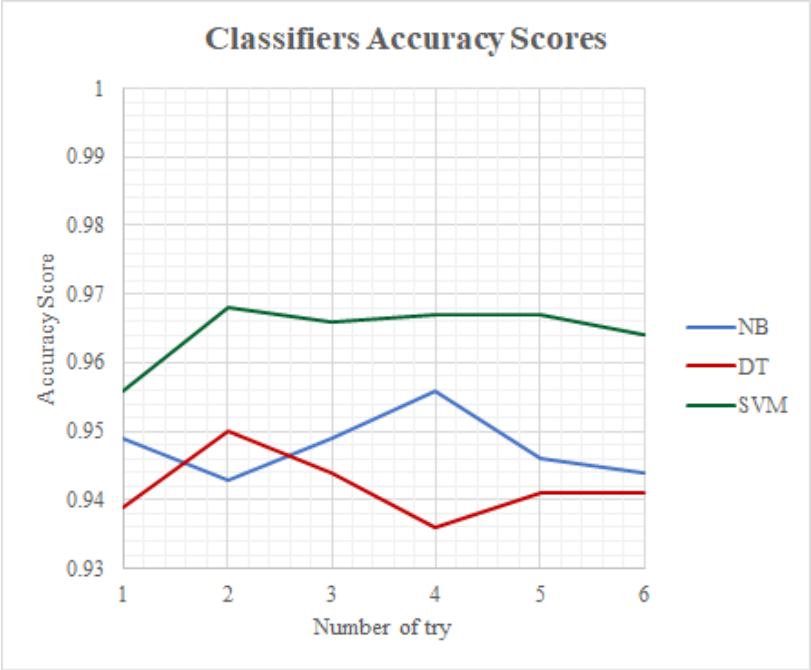

Figure 6.1: Classifiers Accuracy Scores

really well as spam filters on the selected dataset.

6.1.2 Recall

Figure 6.2 shows the Recall scores of each Classifier that were obtained from testing them. Again, it can be seen that SVM Classifier outperforms both NB and DT Classifiers even in predicting the positive instances correctly. NB Classifier seems to have the lowest Recall score among the classifiers. Therefore, as a spam filter, one would prefer to choose the classifier that can find the positive instances (spam emails) well, and has a high accuracy, which in this case is SVM Classifier.

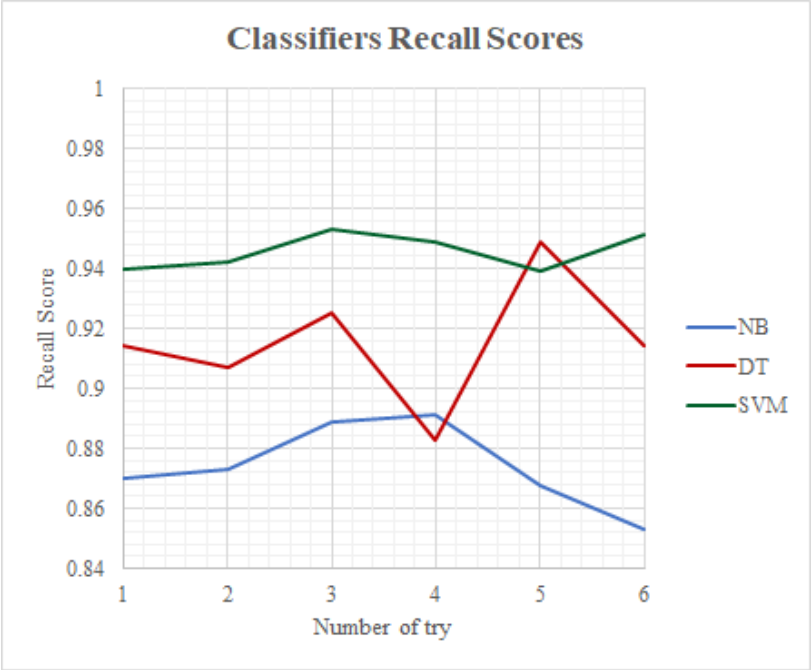

Figure 6.2: Classifiers Recall Scores

6.2 Classifiers Performance in the Presence of an Adversary

Now that each Classifier performance has been evaluated, we can evaluate their performances in the presence of an adversary. Each attack that is implemented in the software system will be mounted on each Classifiers and is presented below.

6.2.1 Feature Deletion Attack

To evaluate the robustness of the classifiers to Feature Deletion attack, all three classifiers have been created and trained on the training set and then tested using the test set that has N features with least weights deleted each time. The values of N were (0, 25, 50, 100, 250, 500, 1000, 1250, 1500). Figure 6.3 presents the performance of each classifier for different values of N features deleted.

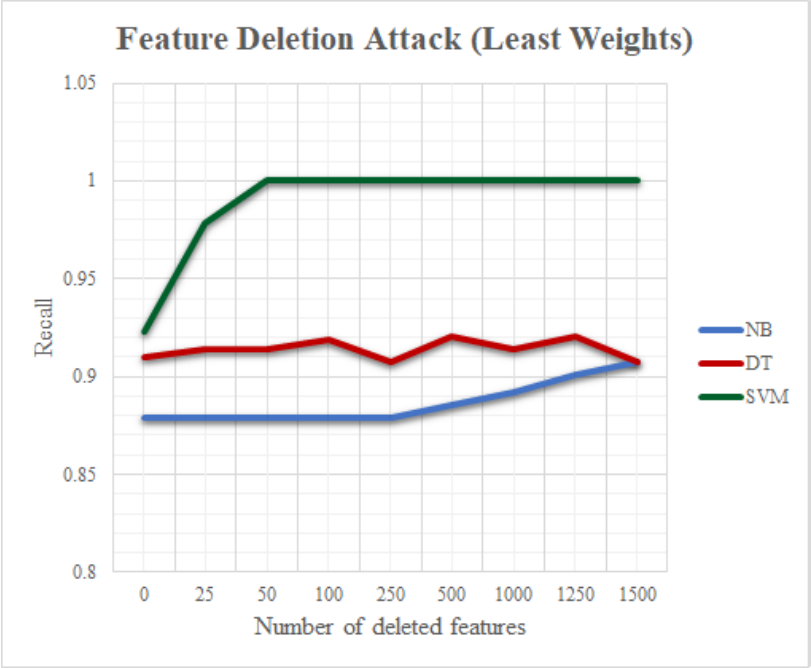

Figure 6.3: Feature Deletion Attack (Least Weights)

It can be seen that all the classifiers seem to perform really good with a high Recall score, even if the number of deleted features increase. Each classifier has an explanation behind these results. Starting with Naïve Bayes classifier, the recall score maintained a high score between 0.87 and 0.90 even though the features with the least weights that were deleted increased each time. This is because the weights given to the features by Naïve Bayes are the probabilities of having them occur in spam emails, therefore, the features with least weights are the ones that have the lowest probabilities and do not affect the decision of classifying an email as spam or not, deleting them then will not change the fact that the emails are spam. Moving on to Decision Tree classifier, which also maintained a high recall score no matter how much features with least weights are deleted. The reason behind this is that Decision tree feature weights are considered as feature importances, the features with the least weights have less importance to make a decision using the Decision tree and are considered irrelevant and do not appear in the representation of tree when constructed. Therefore, deleting them does not change the path followed in the Decision Tree to classify the spam emails. Finally, Support Vector Machine classifier, which has a little different case than Naïve Bayes and Decision Tree classifiers. It can be seen that the recall score is

increasing significantly whenever the number of features with least weights deleted increase. This is because the weights given to the features by Support Vector Machine classifier are divided into two types, negative weights with negative values for the features that occur in ham emails mostly and positive weights with positive values for the features that occur in spam emails. Therefore, when the features that have the least weights are deleted, some data points (spam emails) that were miss-classified as ham because they contain negative weight features will be pushed across the boundary that separates the spam emails from the ham to the spam side, which is the reason behind the recall increase. Overall, it has been seen that all the classifiers are resistant to this attack as no spam emails were miss-classified as ham.

The results from Figure 6.3 raised an interest in investigating Feature Deletion Attack with deleting the features having the highest weights. The same method as deleting the features having the least weight was taken and Figure 6.4 has been created from the results.

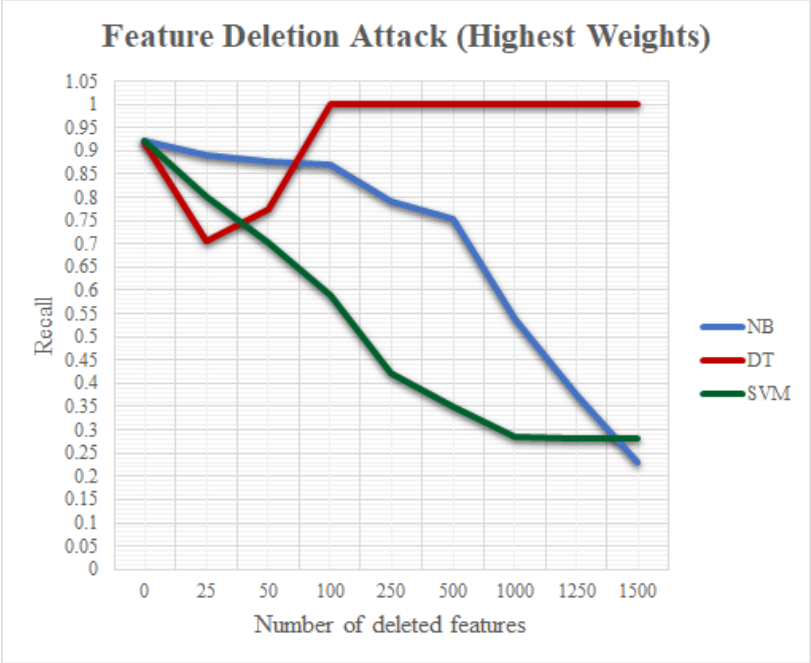

Figure 6.4: Feature Deletion Attack (Highest Weights)

The results in Figure 6.4 clearly support the reasoning behind the results from Figure 6.3 in interpreting the weights of the features in each classifier. It can be seen that both Naïve Bayes and Support Vector Machine Classifiers are suffering from significant loss in their recall score whenever the number of the features having the highest weights are deleted,

since these features have high probabilities of appearing in spam emails for Naïve Bayes Classifier and have high positive weights for Support Vector Machine Classifier and deleting them from the spam emails results in miss-classifying them as ham emails. In the case of Decision Tree Classifier, Figure 6.4 shows that when 25 features having highest weights were deleted, the recall score decreased from 0.9 to 0.7. This is because the features that are most important to classifier's decision has been deleted. So, this case of Feature Deletion attack seems to pose security issues for the classifiers.

6.2.2 Free Range and Restrained Attacks

Free Range attack

To evaluate the robustness of the classifiers against Free Range Attack, each classifier is tested after mounting the attack with having the value of the attack severeness increased each time C_f. Table 6.1 lists the recall scores of the classifiers with the Free Range Attack.

	$C_f = 0$	$C_f = 0.3$	$C_f = 0.7$	$C_f = 1.0$
NB	0.87	0.07	0.03	0.04
DT	0.91	0.07	0.02	0
SVM	0.83	0	0	0

Table 6.1: Classifiers Recall scores in presence of Free Range attack

Clearly, whenever C_f increases, all classifiers suffer more from miss-classifying the spam emails as ham emails. This indicates that these machine learning classification algorithms are vulnerable to attacks that changes the values of input data.

Restrained attack

Looking now at the Restrained attack, a similar approach to Free range attack has been taken to investigate the robustness of the classifiers against it. Each possible combination of the attack severeness value $C_\delta \in \{0.0, 0.3, 0.7, 1.0\}$ with the data movement factor value $C_\xi \in \{0.0, 0.3, 0.7, 1.0\}$ have been tested. Tables 6.2, 6.3 and 6.4 list the recall scores for Naïve Bayes, Decision Tree and Support Vector Machine Classifiers respectively.

NB	$C_\delta = 0$	$C_\delta = 0.3$	$C_\delta = 0.7$	$C_\delta = 1.0$
$C_\xi = 1.0$	0.54	0.59	0.72	0.87
$C_\xi = 0.7$	0.58	0.65	0.76	0.87
$C_\xi = 0.3$	0.71	0.77	0.83	0.87
$C_\xi = 0$	0.87	0.87	0.87	0.87

Table 6.2: Naïve Bayes Classifier Recall scores in presence of Restrained Attack

DT	$C_\delta = 0$	$C_\delta = 0.3$	$C_\delta = 0.7$	$C_\delta = 1.0$
$C_\xi = 1.0$	0.11	0.19	0.36	0.92
$C_\xi = 0.7$	0.18	0.24	0.76	0.87
$C_\xi = 0.3$	0.37	0.45	0.9	0.92
$C_\xi = 0$	0.91	0.90	0.92	0.92

Table 6.3: Decision Tree Classifier Recall scores in presence of Restrained Attack

Since we are interested in the scenario were both attack aggressiveness increases ($C_\delta < 1.0$) and a greater amount of data movement is given ($C_\xi > 0$), we can focus on the recall scores that are lined along the diagonal. It can be seen that whenever the attack aggressiveness and the data movement factor increase, all the three classifiers recall score tend to decrease significantly. But, Naïve Bayes classifier seems to be more robust than Decision Tree and Support Vector Machine classifiers by having 46% of the spam emails miss-classified when Decision Tree classifier miss-classified 89% of the spam emails and Support Vector Machine miss-classified 60% of the spam emails in the worst case ($C_\delta = 0$ & $C_\xi = 1$). So again, we can see that all these machine learning classifications are vulnerable to the attacks that change their input data values. In the case of Naïve Bayes Classifier, changing the values of the input data changes their probabilities, which leads to miss-classifying them. Moreover, in Decision Tree Classifier changing the values leads to following a different path for classifying the spam emails in the constructed Decision Tree. Finally, in Support Vector Machine Classifier changing the values in the spam emails leads to moving the hyperplane (boundary) that separates the classes.

SVM	$C_\delta = 0$	$C_\delta = 0.3$	$C_\delta = 0.7$	$C_\delta = 1.0$
$C_\xi = 1.0$	0.4	0.46	0.63	0.83
$C_\xi = 0.7$	0.46	0.53	0.76	0.83
$C_\xi = 0.3$	0.62	0.68	0.76	0.83
$C_\xi = 0$	0.83	0.83	0.83	0.83

Table 6.4: Support Vector Machine Classifier Recall scores in presence of Restrained Attack

6.2.3 Good Words Attacks

To determine the effectiveness of Good Words attacks against the classifiers, each classifier has been created, trained and then tested in presence of both Good Word Attacks, First N Words and Best N Words. The recall scores of the classifier are presented in Figure 6.5.

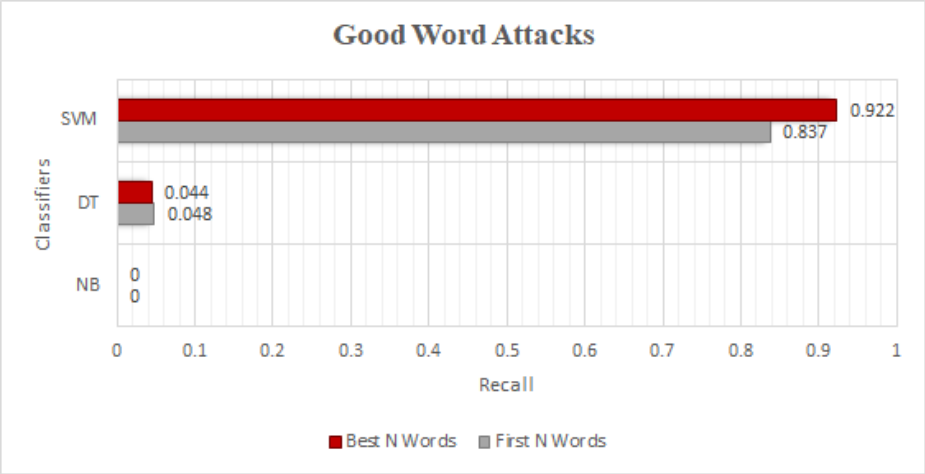

Figure 6.5: Good Word Attacks

Unlike Naïve Bayes and Decision tree Classifiers that are seen to be vulnerable to these type of attacks, Support Vector Machine Classifier is seen to be resistant. While Naïve Bayes miss-classified 100% of the spam emails as ham in both types of the attack, Support Vector Machine classifier miss-classified 8% when attacked using First N Words and 17% when attacked with Best N Words. According to Lowd and Meek, a Classifier is more susceptible to Good Word Attacks if it has features with very negative weights, since they disguise spam more efficiently [13]. Therefore, this means that both Naïve Bayes and Decision Tree classifiers have features with very high negative weights who represent the features that always occur in ham emails (good words), and adding these features to the spam emails

leads to miss-classifying them. But, in Support Vector Machines, features appearing in ham emails seem to have low negative weights. So, adding these words do not change the fact the emails are spam, since they have little or no effect against the positive weight features.

6.3 Black Box Model Identification

Another important aspect that this project aims to prove, is the possibility of identifying the underlying model or classifier of a Black Box model using adversarial examples that were created previously for White Box models. Therefore, two Black Box models were created and trained with an unknown underlying classifier and then attacked with two different attacks, Good Word First N Words attack and Feature Deletion Attack that were created for three existing models, Naïve Bayes , Decision tree and Support Vector Machine classifiers. The reason why these attacks were chosen, is because they depend heavily on the underlying model to find the best way to fool the classifiers and therefore, the resulting adversarial example will differ for each classifier. After attacking the Black box models, each model has been tested. The evaluation metric that has been focused on is the Recall score, since the goal of the adversaries is to make the model miss-classify the spam emails. Each Black Box recall scores are presented in two separate tables, table 6.5 and 6.6 that are presented below.

Adversarial example designed for	Good Word Attack	Feature Deletion Attack
NB	0	0.57
DT	0.84	0.86
SVM	0.8	0.62

Table 6.5: Black Box 1 Recall scores

Adversarial example designed for	Good Word Attack	Feature Deletion Attack
NB	0	0.91
DT	0.88	0.96
SVM	0.83	0.28

Table 6.6: Black Box 2 Recall scores

From table 6.5, it can be clearly seen that the attacks that were designed for Naïve Bayes classifier decreased the recall score of the Black Box significantly and the attacks that were designed for Support Vector machine and Decision tree classifier do not seem like they are

affecting the recall score of the Black Box. Thus, the underlying model of the first Black box could be Naïve Bayes. After checking the ground truth value for the underlying algorithm for Black Box 1, it was Naïve Bayes. For the other Black Box, table 6.6 shows that both attacks designed for Naïve Bayes and Support Vector machine Classifiers affect the Black Box. But, only the Good Word Attack from Naïve Bayes seems to be able to fool the Black box and the Feature Deletion Attack seems to have no effect on it. So we can exclude it and look at the Support Vector Machine Classifier attacks. While Good Word Attack seems to have no effect on the Black Box, Feature Deletion attack reduced the Recall significantly. Since we already know that Support Vector Machine Classifier tends to be resistant from Good Word attacks that are designed for it and Feature Deletion attacks can easily fool it, we can say that the second Black Box underlying model is a Support Vector Machine Classifier. After checking the ground truth value of the underlying algorithm for Black Box 2, it turned out to be Support Vector Machine.

So, these results made it clear that it is possible to identify the underlying models of Black Boxes using adversarial examples designed for other classifiers. Another interesting point from the results is the transferability of adversarial examples. We saw that even though the second Black Box underlying model was Support Vector Machines it got affected by the Good Word adversarial example created for Naïve Bayes. This proves that adversarial examples have a transeferability property that was discussed in Chapter 2.

Chapter 7

Future Work

"Everything takes longer than you think, even when you take into account Hofstadter's Law"

(Douglas Hofstadter)

Due to the limited time allowed for this project, there are several aspects of the software system that could potentially be improved in the future. The first aspect would be the addition of classifiers such as, Neural Networks, Random Forest and Nearest Neighbour algorithms and the addition of more attacks such as, Binary and Coordinate Greedy attacks and Cost Sensitive Attack. This would help in further investigating and examining the security of machine learning classifiers in the presence of an adversary. Moreover, another aspect that could be improved would be allowing the users to change the parameters of the attacks such as, the number of deleted features from the interface instead of changing them from the code itself and then running the whole software again. Another desirable improvement is enhancing the user interface of the software system. This could be done using web technologies such as, HTML, CSS combined with Java script to create a graphical user interface that will make the software usable even by beginners. The graphical interface could allow presenting graphs from the results of testing or attacking the classifiers and allow choosing parameters for the classifiers and attacks.

Chapter 8

Conclusion

This project was set out to investigate the security of spam filters that are built upon Machine Learning Classification algorithms against adversarial attacks and examine the possibility of identifying the underlying algorithm of a Black Box model using existing adversarial examples designed for other models. In order to achieve this, a software system has been developed to allow modelling the combat between classifiers representing spam filters and adversarial attacks during the test time of the classifiers. The main findings of this project suggest that spam filters built using machine learning are not safe enough against adversaries that can change the data values of their input during testing. In addition, Black Boxes underlying model could be easily identified by creating adversarial examples for other known models and using them to attack the Black box models. This means that spam filters should not be built using machine learning classification algorithms since they can be defeated easily.

Chapter 9

Reflection on Learning

The greatest lesson that was learnt from doing this project was the importance of writing the report as you go along. The approach that I took was different, when it came to the implementation phase I implemented the whole software system in three weeks and then wrote the report. This made me struggle with writing the report, because the ideas were not fresh anymore, and I had to go back over things I already did 3 weeks ago, which wasted so much time. Another lesson that was learnt is to always use LaTeX for writing academic reports. I have studied LaTeX a few years ago and had a little experience with it and never thought of using it for my reports, but in this report, things have changed after trying to write the report using Microsoft Word and saw how it requires too much effort to produce an organised report with figures and mathematical equations. Thus, utilising LaTeX for writing the report made the report look more organised and professional with great ease. Therefore, LaTeX has been an invaluable took and will be taken in considerations for future reports. On the other side, a great knowledge was gained about Machine Learning, which I had minimal knowledge about and Adversarial Machine learning, which I clearly did not know that this field exists while doing the background research. This knowledge gave me a great insight into how Machine learning works and used for spam filters and how to evaluate their security. This gave me the interest to explore the security aspect in the field of machine learning more deeply not only in the case of spam filters. Finally, I learnt that when creating a software, it is always a good approach to break the code into parts, which consists of classes and modules instead of coding everything in one file. This made the code more organised, reusable and easier to debug. Overall, despite the fact that this project lasted only for few months, I believe that the knowledge and skills I gained will be beneficial in my future life.

References

[1] Alpaydin, E. (2010). *Introduction to Machine Learning.* [Online]. Available at: http://cs.du.edu/~mitchell/mario_books/Introduction_to_Machine_Learning_-_2e_-_Ethem_Alpaydin.pdf [Accessed: 05-03-2018].

[2] Bhowmick, A. et al. (2016). *Machine Learning for E-mail Spam Filtering: Review, Techniques and Trends.* [Online]. Available at: https://arxiv.org/pdf/1606.01042.pdf [Accessed: 02-04-2018].

[3] Globerson, A. et al. (2006). *Nightmare at Test Time: Robust Learning by Feature Deletion* . [Online]. Available at: https://www.utdallas.edu/~muratk/courses/dmsec_files/robust_icml06.pdf [Accessed: 05-04-2018].

[4] Grosse, K. et al. (2017). *On the (Statistical) Detection of Adversarial Examples* . [Online]. Available at: https://arxiv.org/pdf/1702.06280.pdf [Accessed: 02-03-2018].

[5] Gunopulos, D. et al. (2003). *Scaling up the Naive Bayesian Classifier: Using Decision Trees for Feature Selection* . [Online]. Available at: http://alumni.cs.ucr.edu/~ratana/DCAP02.pdf [Accessed: 11-03-2018].

[6] Guzella, T. et al. (2009). *A review of machine learning approaches to Spam filtering.* [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.455.7614&rep=rep1&type=pdf [Accessed: 03-03-2018].

[7] Han, J. et al. (2012). *Data Mining Concepts and Techniques.* Elsevier.

[8] IBM-Security (2017). *IBM X-Force Threat Intelligence Index 2017.* [Online]. Available at: https://www.ibm.com/security/data-breach/threat-intelligence [Accessed: 31-01-2018].

[9] Jurafsky, D. et al. (2007). *Naive Bayes and Sentiment Classification* . [Online]. Available at: https://web.stanford.edu/~jurafsky/slp3/6.pdf [Accessed: 04-03-2018].

[10] Kantardzic, M. (2011). *DATA MINING Concepts, Models, Methods, and Algorithms.* [Online]. Available at: https://doc.lagout.org/Others/Data%20Mining/Data%20Mining_%20Concepts%2C%20Models%2C%20Methods%2C%20and%20Algorithms%20%282nd%20ed.%29%20%5BKantardzic%202011-08-16%5D.pdf [Accessed: 03-03-2018].

[11] Keras (2017). [Online]. Available at: https://keras.io/ [Accessed: 20-03-2018].

[12] Kurakin, A. et al. (2017). *ADVERSARIAL MACHINE LEARNING AT SCALE* . [Online]. Available at: https://arxiv.org/pdf/1611.01236.pdf [Accessed: 04-03-2018].

[13] Lowd, D. et al. *Good Word Attacks on Statistical Spam Filters* . [Online]. Available at: https://www.utdallas.edu/~muratk/courses/dmsec_files/125.pdf [Accessed: 29-03-2018].

[14] Metsis, V. et al. (2006). *Spam Filtering with Naive Bayes – Which Naive Bayes?* . [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.5542&rep=rep1&type=pdf [Accessed: 10-03-2018].

[15] Saini, U. (2008). *Machine Learning in the Presence of an Adversary: Attacking and Defending the SpamBayes Spam Filter.* [Online]. Available at: https://people.eecs.berkeley.edu/~adj/publications/paper-files/EECS-2008-62.pdf [Accessed: 31-01-2018].

[16] Scikit-learn (2017). [Online]. Available at: http://scikit-learn.org/stable/index.html [Accessed: 04-03-2018].

[17] Shwartz, S. et al. (2014). *Understanding Machine Learning: From Theory to Algorithms.* [Online]. Available at: https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/understanding-machine-learning-theory-algorithms.pdf [Accessed: 07-03-2018].

[18] Theano (2017). [Online]. Available at: http://deeplearning.net/software/theano/ [Accessed: 20-03-2018].

[19] Westra, E. (2016). *Modular Programming with Python Introducing modular techniques for building sophisticated programs using Python.* [Online]. Avail-

able at: http://file.allitebooks.com/20161026/Modular%20Programming%20with%20Python.pdf [Accessed: 28-04-2018].

[20] Witten, I. et al. (2016). *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier.

[21] Zhou, Y. et al. (2012). *Adversarial Support Vector Machine Learning* . [Online]. Available at: https://www.utdallas.edu/~muratk/publications/kdd2012.pdf [Accessed: 02-04-2018].