Differential IP Clustering for Vulnerability Detection using applied Machine Learning

Sebastian Bocquier
Division of Computing and Mathematics
University of Abertay Dundee

A thesis submitted for the degree of
Honours of Science
in
Ethical Hacking and Computer Security

April 22, 2017

This thesis is dedicated to
no one
for no special reason

Acknowledgements

I would like to take this opportunity to thank my project supervisor, Dr Xavier Bellekens, who introduced me to the topic of Machine Learning and offered guidance from the beginning through to the completion of this project. Without whom this project would not have been possible. I would also like to thank the module leader, Colin McLean, who has given me continuous advice throughout my years of study at Abertay University.

Abstract

plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle.

Contents

List of Figures

Chapter 1

Introduction

This section will give a brief background and history of Machine Learning before going in to depth with the definitions and types of algorithms used in modern Machine Learning systems. An analyses of the impact Machine Learning has in the field of computer security and how it can be applied directly to penetration testing and red teaming. Research aims and objectives will be provided as well as a statement to the structure of the rest entire thesis.

1.1 Background

The term machine learning was first defined by Arthur Samuel in the year 1959 as a "Field of study that gives computers the ability to learn without being explicitly programmed" and later as "a field of study that concentrates on induction algorithms and on other algorithms that can be said to 'learn'", Kohavi & Provost (1998). Machine learning has largely evolved from several subfields of artificial intelligence, specifically, computational learning and pattern recognition but now it stands on its own with its subfield Deep Learning being at the forefront of technology. Machine learning consists of the studying and construction of algorithms that can learn from and make predictions of data, Simon (2013). The early information available for machine learning was almost entirely theoretical due to the lack of processing power at the time. One of the early pioneers was Valiant (1984) whom developed a PAC learning framework and established the theory of a learnable algorithm, Munoz (2012). Modern machine learning algorithms use many calculations from statistics, information theory, theory of algorithms, probability and functional analyses, Munoz (2012).

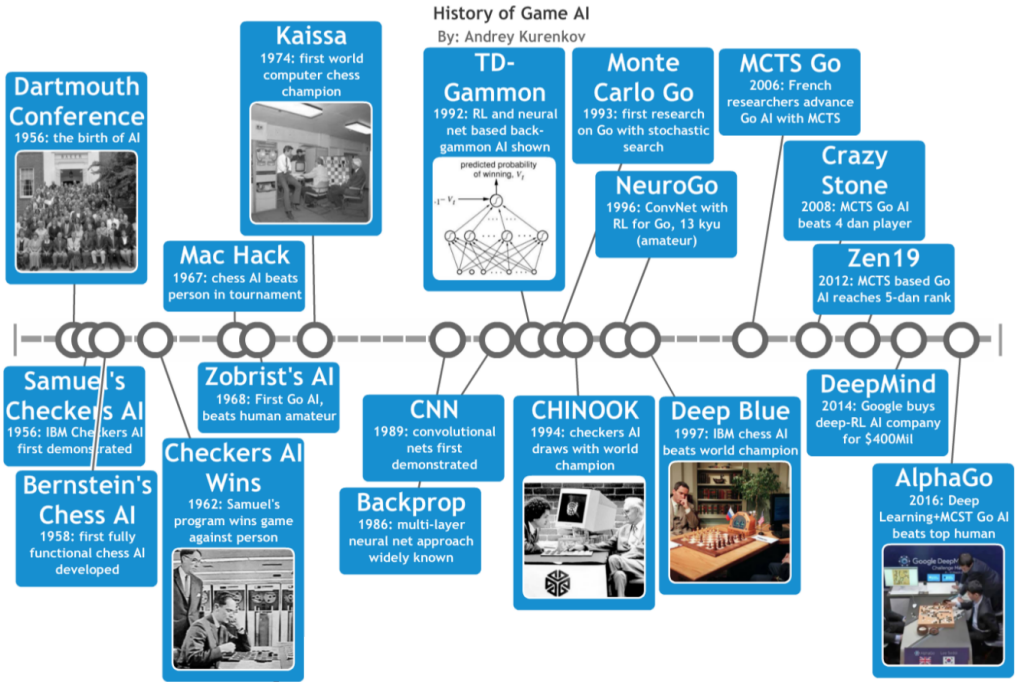


Figure 1.1: Timeline of artificial intelligence for games, Kurenkov (2016)

1.2 Current State of Machine Learning

In the recent years, machine learning accuracy and efficiency has increased dramatically which has caused the field to rapidly gain in popularity, especially in deep learning. One of the major reasons for this is due to the improvement in graphics processing units which in some computational tasks can outperform central processing units by an order of magnitude or more, Mattsson (2016). The rapidly growing field of deep learning could be made apparent by Googles recent self-driving car and LeNet image recognition system, however, examples of machine learning can be found in just about every field with a classification or regression problem. Some of which include: natural language processing using morphological analyses for linguistics (Black & Zernik (1994)), document classification and spam detection, better image recognition than humans (Karpathy (2011)) and beating a professional player at the complex Chinese game Go (DeepMind (2016)). Figure 1.1 illustrates the advancement of machine learning using the context of game artificial intelligence players.

The majority of projects over the past few years have focused on Deep Learning due to its performance and versatility with complex problems but also because of the media coverage with large scale projects by international corporations such as Google and Microsoft. The idea of deep learning has been around since the 1980s where a

Japanese scientist, Kunihiko Fukushima, proposed a hierarchical, multi-layered artificial neural network named Neocognitron, designed for handwritten character recognition. Neocognitron was recognised as the inspiration for convolutional neural networks, the most commonly used deep learning algorithm, LeCun et al. (2015). Deep learning attempts to model complex abstractions in data by using a multiple-level architecture most commonly compromising of artificial neural networks and non-linear transformations in its algorithms, Mattsson (2016). As such, the concepts of deep learning are extensions of regular machine learning algorithms.

In the past year a new topic has become of consider interest in the machine learning field, Automated Machine Learning, which can be considered to cover the tasks of algorithm selection, hyperparameter tuning, iterative modelling and model assessment, Mayo (2017). By the end of 2016 the python Auto-sklearn library was created based off of the scikit-learn library which encompasses these tasks, created by a team from the University of Freiburg it won the KDnuggets AutoML challenge, Matthias Feurer (2016). Using these examples, its possible to hypothesize that the future of machine learning will include automatic deep learning, however, these individual fields of machine learning are still in their infancy and far from being used together.

The computer security industry due to its nature, has many examples of machine learning implementations such as intrusion detection systems using machine learning or deep learning for anomaly detection. One of which has been commercialised under the company name Deep Instinct and advertises zero-day detection using deep learning. However, the majority are very similar and are all blue team based security solutions. Researching into Red team or penetration testing tools using machine learning resulted in a disappointing lack of tools or ideas considering the vast amount available for Blue team.

The sole documented research found for red team was for an automatic penetration testing project named Auto Red Team (ART) framework by Lu, Song of Iowa State University in 2008 which used decision trees and hard programmed exploits. This meant the entire hard programmed exploit section had to be reconstructed for each use case, this would not be ideal but also extremely time consuming. Further analyses of the ART framework can be found in the literature review of "Auto Red Team: a network attack automation framework based on decision tree". There have also been tools and libraries created to test the security of software which use machine learning

models such as Deep Pwning. Deep Pwning is an open source metasploit plugin which allows the tricking of machine learning models. This field of research was named Adversarial Machine learning and the first paper of which was respectably named "Adversarial Machine Learning" and published by ACM in 2011.

There is an extensive amount branches to machine learning and unfortunately, too many to cover in this thesis due to time constraints. Therefore, this introduction will only detail the most popular types of machines learning algorithms.

1.3 Significance of study

The increased demand for penetration testers justifies the need for more effective and advanced tools to conduct their security assessments. The goal of a penetration test being to find vulnerabilities in a system using techniques similar to that of a malicious hacker. This means malicious hackers will continue to use the most advanced methods to gain access to critical systems and thus security teams must also continue to advance their toolset to be as effective and efficient as possible. Penetration tests can last a time scale of anything between one day to several months and more advanced machine learning tools would allow for a more efficient use of this time. The large variety of machine learning models used in the blue team results in a large variety of models required to test them using adversarial machine learning as well as advanced machine learning tools, specifically for the red team in order to bring each team to the same level. Having both red team and blue team on the same level is beneficial for the industry as a whole, providing competition between both sides and to continue to strive for improvement.

This project aims to help correct this unbalance by designing and developing a proof of concept red team tool using machine learning techniques.

The application must be able to be used as an aid to a security professional during a security assessment or capture the flag hacking events in order to be classed as a red teaming tool.

This application must also be scalable and versatile to be used in varying sized network environments.

The tool will be critically analysed and recommendations of related future work will be provided.

1.4 Types of Machine learning

As mentioned above there are many different machine learning algorithms available. Each algorithm can be classed based on the problem, required output and several factors of the data set, such as whether it includes labels and the amount of values it includes. The two primary problems machine learning algorithms provide solutions to, can be put in to two categories which are not mutually exclusive and can be combined in certain use cases. These are classification and regression problems. The following describes and states the differences of these types.

1.4.1 Classification

With the growth of big data, unstructured data is more prevalent than its structured counterpart. This is because, "while the amount of structured data has grown fast, the amount of unstructured data has grown much faster" Simon (2013). This creates the need for ever more efficient data analytics and is a typical classification problem for machine learning. There are many types of classification, simple types such as a linear classifier and then more complex types such as multiclass and structured classifiers. Classification is largely used in data mining and statistical analyses for these purposes. In short, classification is used when you require an input variable to be identified as part of a group or label, resulting in the full dataset being categorised. Classification can only take a finite set of values such as picking from 1 of N values. In classification, each incorrect answer is equally incorrect, compared to regression where incorrect answer can be varying levels of incorrect.

1.4.2 Regression

Regression problems are for when prediction of real continuous values is required, such as predicting stock market values and detecting the age of a person from a picture, Rossant (2014). Regression analyses involves predicting and estimating a response based on previous data and input variables. As mentioned above regression answers can have a varying level of inaccuracy as appose to binary correct or false predictions. This is due to regression using continuous values. Figure 1.2 provides a basic illustration example of these two problems.

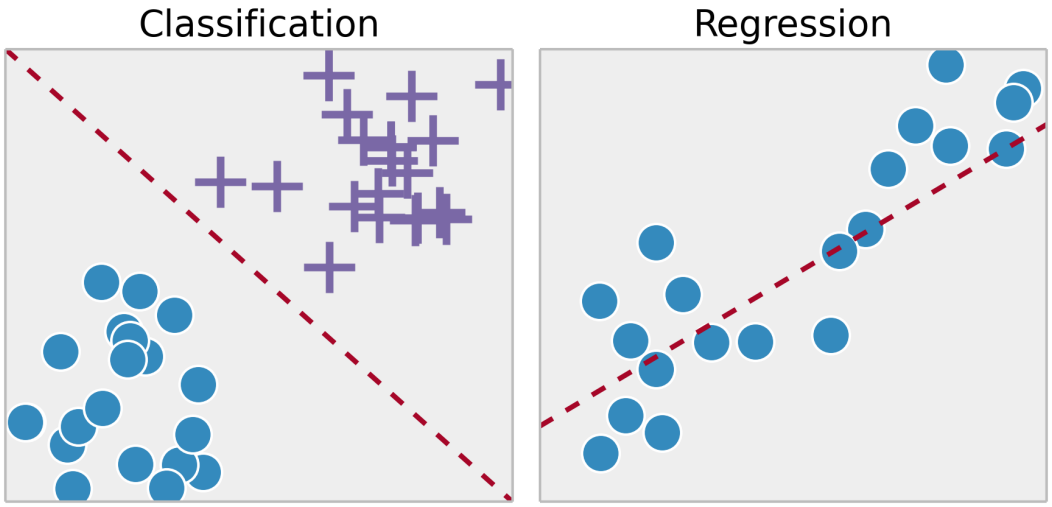


Figure 1.2: Illustration of machine learning types, Rossant (2014)

1.5 Types of Learning algorithms

The most commonly found learning methods for algorithms are supervised and unsupervised followed by semi supervised and reinforcement learning which are also fairy common. These are explained briefly as follows.

1.5.1 Supervised

In a supervised training model, the entire data-set used for training is pre-labelled so that the algorithm can use pattern analyses to predict given values after training. A use case scenario for supervised learning includes the above stock market example where current values with labels are given and the model is used to predict future labels.

1.5.2 Semisupervised

Due to the expensive nature of labelled data in most cases, semisupervised learning uses a split of labelled and unlabelled data to train the model. This is most often split unevenly with the large majority of training data being unlabelled. This model is often used in cases where the cost of labelled data-sets is simply too expensive.

1.5.3 Unsupervised

The opposite of supervised learning, where the training data-set has no labels and the model must attempt to determine the correct answer itself. This is often used as a method to determine a structure in the data given. These models can identify segments of similar attributes such as clustering. Unsupervised learning is the primary method used for this thesiss methodology.

1.5.4 Reinforcement

Reinforcement learning works similarly to heuristic algorithms in the sense that every possibility is attempted and assigned a score in which the iteration with the highest score is used as the model output. Whilst training this model the algorithm uses the highest scores iteration to modify its calculations for greater accuracy and efficiency. This model is often used in robotics as well as game design for path navigation calculation and computer player AI (artificial intelligence). The following is an example use case: A chess player AI calculates each possible move it can make using a reinforcement model for each of its turns. The model assigns a score to each move it could possible make at that point in time and weights them based on pieces acquired, future strategy prospects and defence risk. Similar models have been used for AIs mentioned in Figure 1.1.

1.6 Common Machine Learning Algorithms

As mentioned above, due to the amount of machine learning algorithms available and project time constraints, only a few of the most common algorithms will be described below.

1.6.1 Linear Regression

Linear regression predicts real values based off continuous data of two variables. It does this by using a best fit or regression line over the existing data extending in predictions. If the data does not indicate any positive or negative trends then using linear regression will likely not be a very useful model. The trend or direction of data can be calculated using correlation coefficient as follows, where $(x_2, y_1), (x_n, y_n)$ is the observed data.

$r = \frac{1}{n-1}\Sigma \left(\frac{x-\bar{x}}{s_x}\right)\left(\frac{y-\bar{y}}{s_y}\right)$

A value that is close to 1 would indicate positive correlation where -1 would indicate negative correlation. A normalised covariance calculation may also be used instead.

1.6.2 K-Nearest Neighbours

K-nearest neighbours (or KNN) is a widely used computationally expensive supervised learning model for data classification but can also be used for regression problems. The value k refers to the distance is which to weigh the number of class nodes inside. Several distance functions can be used such as Euclidean, Manhattan and Minkowski with the most common being Euclidean. Euclidian distance is the straight-line distance between the two nodes. On a two-dimensional plane it is measured using the following formula where the coordinates are $\mathbf{p} = (p1,\ p2)$ and $\mathbf{q} = (q1,\ q2)$.

$d(p, 1) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}$

For e.g. if k = 1 then the current node will be assigned to the same class as its nearest node, where as if k = 5 then the class with the largest number of nodes in a distance of 5 will be selected to represent the current node. KNN is not resistant to bias in data-sets and thus data must be normalised before inputted into the model. KNN models are not to be confused with K-means clustering models as they share a very loose relationship being that KNN can be used to add data into pre-existing K-means clusters known as a nearest centroid classifier.

1.6.3 K-means Clustering

K-means Clustering is an unsupervised model in which the number of clusters is specified in advance. K being the number of clusters the model will output for the given data. The value for K can be assigned manually or the optimal value can be calculated by using either the elbow method or the gap statistic, the latter of which is used and explained in the methodology section of this thesis with a practical use of K-means. The K-means algorithm itself is also known as Lloyd's algorithm and uses iterative refinement to determine the best clustering.

1.6.4 Dimensionality Reduction

Dimensionality reduction algorithms are exactly what the name suggests. With the dramatic increase in variable types and raw data size being captured leading to the invention of the term Big Data, datasets have become very large. These large datasets have become the main bottleneck for machine learning performance,

especially in computationally expensive models such as KNNs and K-means. There needs to be a way to identify the significance of each variable in the data-set in order to give weights to their values for use in calculations. Principle Component Analyses (PCA) can be used for this purpose as well as for variance maximization. The PCA algorithm attempts to detect correlation between each of these variable types and uses this information to detect the vector directions of maximum variance in high-dimensional data. The algorithm then projects these maximum variance vectors into a smaller dimension whilst attempting to keep the most amount of information and greatest variance as possible. If the measurement scales of the dataset variables are not equal, then the data should be normalised prior to PCA due to the variance maximization functions of the algorithm.

1.6.5 Decision Trees

Decision trees are a classification model that uses supervised learning to calculate either categorical or continuous dependent variables. Decision trees can also be designed manually as was traditionally done. As the name suggests, the model works by splitting the data at each branch of the tree by asking questions and making decisions on each layer. The concept of decision trees is simple however they can be combined together, or include other machine learning models within its layers greatly increasing the complexity of this model.

1.6.6 Random Forests

Random decision forest is one of these complications where the model contains several decision trees built during the training stage and can also be used with unsupervised learning. These random decision forests can be used for classification or regression by simply using the mode of the results from the forest or the mean in the case of a prediction problem. They creation of this model differs from regular decision trees as the training algorithm applies bootstrap aggregating, commonly referred to as bagging, followed by feature bagging resulting in a random decision forest. Bagging increases the stability and accuracy of decision trees by reducing variance and avoiding the overfitting of data. It does this by generating additional sets data based on the given data-set which results in a larger data-set overall reducing the variance but increasing the predictivity of the model. Bagging can also be used on several other machine learning algorithms. Feature bagging also known as the random subspace method, selects random samples from the entire dataset in order to reduce the

correlation between each tree in the forest. This is done to produce varied models for each tree as appose to them being very similar if they were all trained on the entire data-set. Random forests can be complicated once more by adding a neural network within its layers, or even a deep learning model such as recurrent neural networks.

1.6.7 Aritificial Neural Networks

Artificial neural networks (or ANNs) are commonly used as the bases for most deep learning algorithms such as convolutional and recurrent networks and are loosely based on the way a human brain functions. They can be used with several types of learning methods. Neural networks consist of interconnected neurons on a layered scheme in which every neuron has a function and its output can be seen by its connected neurons to then use it in their own functions. Each neural connection has an assigned weight for its output which is calculated during the training stage of the model. The layers consist of three types, an input, an output and hidden layers sandwiched between the two. The number of hidden layers in a neural network is what defines whether it is a deep learning model or not. Models with more than one hidden layer are referred to as deep learning models. How the layers are connected and the way data travels between the layers defines the type of neural network. For e.g. a recurrent neural network allows for sequences to be used for input and output unlike convolutional which can only use fixed values for these. The Google subsidiary DeepMind uses recurrent neural networks in its algorithms to create its AlphaGo AI mentioned previously.

Chapter 2

Literature Review

Chapter 3

Methodology

This chapter contains an in-depth analysis of the design and implementation stages carried out during this project. Initially, the design and goals of the application will be enumerated. Subsequently an analysis of the applications infrastructure will be presented followed by an in-depth detail of the modules and submodules within the applications programming. A test case scenario will be defined and executed to provide a proof of concept. The efficiency and accuracy among other factors based on the test scenario will be analysed during the discussion chapter of this thesis.

3.1 Design

The application designs primary goal was to be able to detect vulnerable machines on a large-scale network infrastructure regardless of topology or host types by using machine learning techniques and automated tool outputs. However, there are several requirements the application must adhere to for it to be a viable tool during a security assessment. This application was created strictly on a proof of concept bases.

The application was designed with the following requirements in mind:

Text progress output with multiple verbosity settings allowing for an experienced tester to understand what the application is doing at any point in time during execution. This is critical as tools used during an assessment on live networks must not hinder or damage the network or its hosts in any way as to disrupt an organisations business.

Several input type parameters for which the tester can utilise based on the current information known about the network. Such as, only using one type of scan file and manually selecting the clustering model.

Several output options including visually in the form of graphs and to a dot type file to be used with other industry applications and reports.

Manual overriding of variables via parameters in order to allow for the application to be scripted and modified by the tester. This will increase the efficiency of using the tool and provide advanced customisation of the algorithms within the application.

Highly versatile with working conditions and configurability. The programming of the application to be highly documented allowing a tester to fix and modify the application code to suit the operations needs. By using a primarily interpreted language as appose to compiled one would allow for this, as well as making the application portable without extra code. For these purposes, the Python programming language was chosen. With the majority of modern tools and scripts used by penetration testers haven been written in Python due to its versatility, reliability and portability, it further enforces this choice.

3.1.1 Application Brief

The application requires several parameters to run and has three different global modes; manual, assisted and automatic. These modes can either be run with Nmap, Nessus or both inputs with the majority of the use case scenarios requiring both. The application will then parse these inputs into labels and features, process the data in several ways and cluster the information based on feature similarities (explained further in section 3.2.4). The text output is then displayed which includes the full details of each cluster within the clustering and several statistics such as adjusted silhouette values (more information found in section 3.2.5 and 3.2.6). When using Dual input mode (both Nessus and Nmap files) the application will combine the data from each and subtract the large similarity clusters. By doing this, the application will determine the most unique hosts within the topology and display them in a new clustering. These most unique hosts, based on probability, will be the most vulnerable on the network and should be prioritised by a tester during the manual security assessment, because it is highly unlikely that the large clusters removed

would contain addresses which are solely vulnerable to the same exploit. This is due to the clustering model prioritising the vulnerabilities that each scanner detects, then appending them to the pre-existing host set, thus rendering that host more unique with a greater feature difference than the other hosts without this vulnerability. The application then renders a graphing interface to display the information as such in described in 3.8.

An example output of the application when using **dual input automatic mode** can be found in text form using maximum verbose level at Appendix B.1.1 and a Figure of the graphing interface GUI at Appendix B.1.2. The data-set used for these examples were Nessus and Nmap scan XML outputs generated from a fictional network. Due to the sensitive nature of the data included within these scans such as SSH keys and vulnerability codes, there are no publicly available data-sets.

The difference between modes and parameters will be explained further in the next section, the *infrastructure analysis.*

3.2 Infrastructure

Figure 3.1 shows the applications primary class infrastructure when in automatic mode. The infrastructure diagram has been created using standard flow diagram symbols to provide understanding of the process types. The application has been programmed for python version 2.7 interpreters and therefore will not have complete functionality without modification for python 3.0 and above. Due to time constraints placed upon this thesis the library NMAP-Cluster, Blackhat USA (2016), was used to conserve time.

In order to execute this application, it is important to have the correct library dependences. This is done via the python package manager PIP and a requirements file, found in Appendix A.1, by executing the command:

```
Pip install -r requirements.txt
```

The following sections include detailed descriptions of the processes symbolised within the infrastructure Figure 3.1. Beginning from the start circle and ending at the display modules.

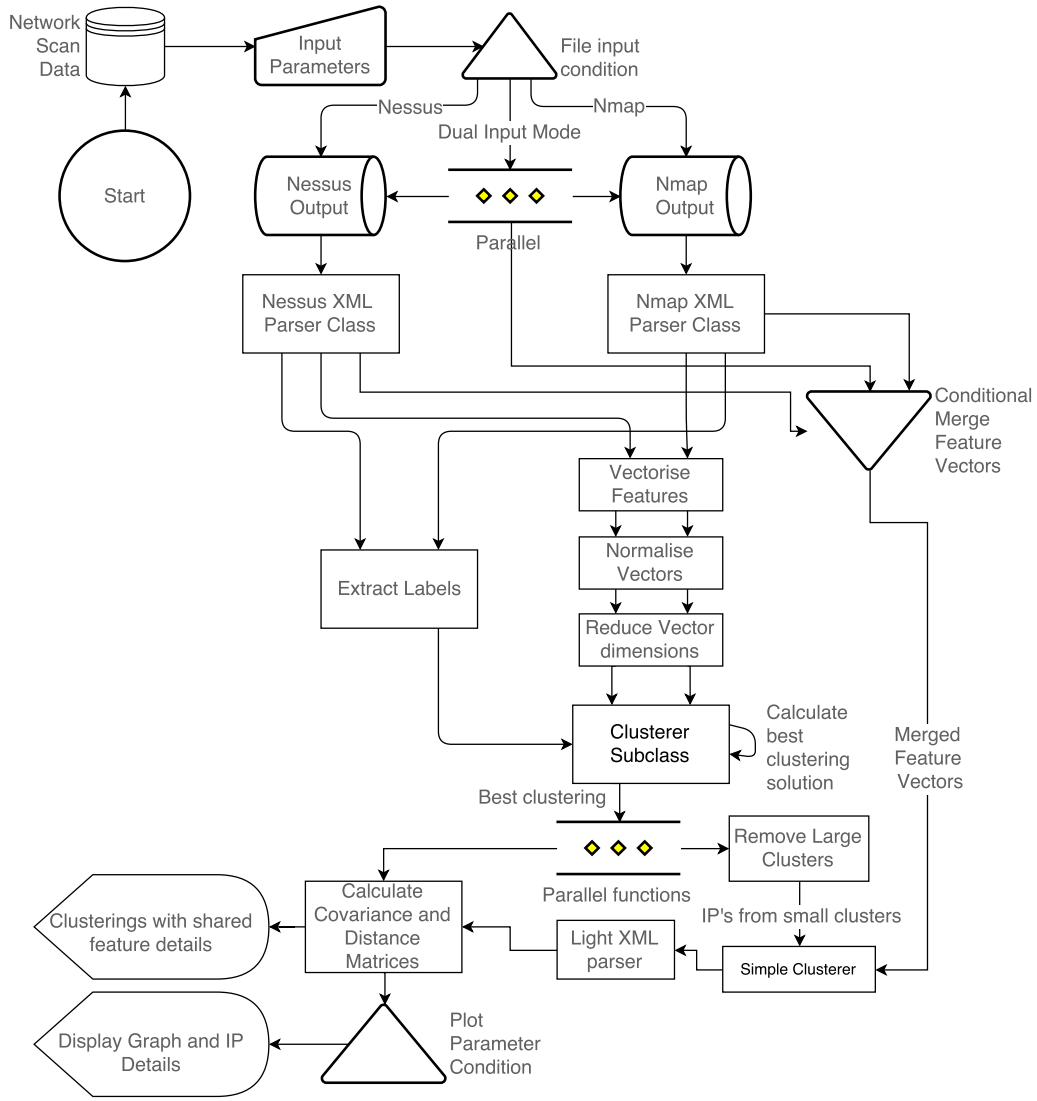


Figure 3.1: Application infrastructure data flow diagram

3.2.1 Usage and Parameters

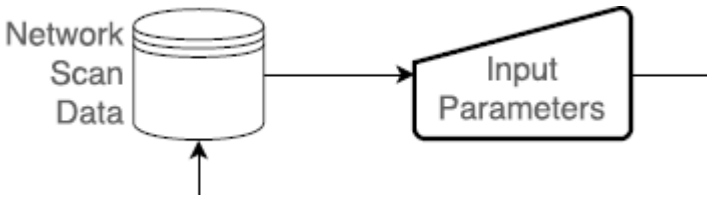


Figure 3.2: Network scan data and input parameter modules from infrastructure diagram at Figure 3.1

The modules shown in Figure 3.2 are those used to store the network scan data and process the applications input parameters. The Network Scan Data module refers

to the XML output of an Nmap scan, Nessus proprietary export file or both. These must be in their respectable formats in order for the parser to recognise them. The files must also be referenced in their correct positional arguments when executing the application such as mentioned in the application usage in Appendix A.2.

The Parameters manual input module on Figure 3.2 refers to the parameters that the application requires to select the correct run configuration. This is required because the application has no execution graphical user interface (GUI), the lack of which was decided for several reasons. Such as allowing for scripting, verbose output and terminal pipe operation commands. This type of interface is generally preferred by professionals due to the speed and reliability it provides over a standard GUI. The graphing stage of the application does however, provide a GUI to allow for manipulation of the graphs in multiple ways. The application usage found in Appendix A.2 includes a full description of the possible parameters. The parameters are passed into the data processing section of the application explained in the next session. The three possible run modes are explained in a later section 3.6.

3.2.2 Data Processing

The data processing modules from the infrastructure diagram have been highlighted in Figure 3.3 and will be described in the following paragraphs.

Depending on the files specified within the input parameters, the application will either feed the Nessus, Nmap or both XML files into the parser classes. These parser classes will take the data from each file and transform it into IP addresses and features which are then passed individually into vectorizers. Vectorization is required for the features to be understood by the clustering algorithm. Vectorization refers to the general process of turning a collection of text (in this case machine attributes) into numerical feature vectors as float values. It is important to note that data from each scanner file is kept separate until the final process. The vectorization class is short and concise due to it having only two purposes, to call the parsers and vectorise the returned results. More information on the vectorization of each file as well as the raw python class code can be found in Appendix A.4.

Once the data has been vectorised it must be normalised in order to avoid large value bias when using dimensionality reduction algorithms such as PCA. Data normalization scales the values to within the same range whilst keeping the data variance

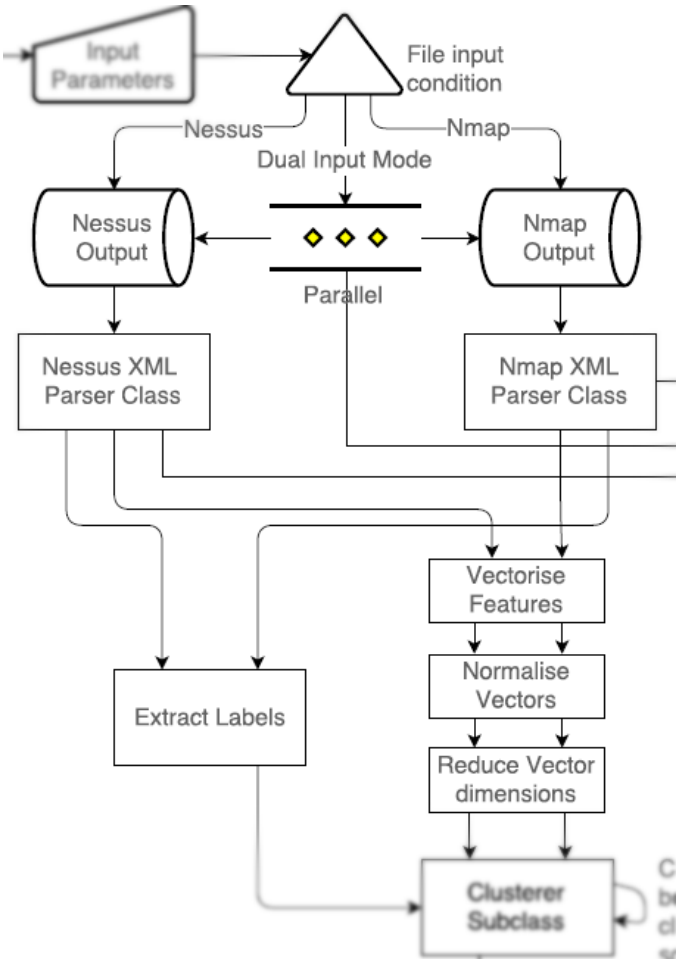


Figure 3.3: The data processing section from the infrastructure diagram at Figure 3.1 with the irrelevant modules blurred out

and eliminating the bias problem. This is programmed immediately after vectorisation in the applications initialisation class found in its raw code form at Appendix A.3.

The major difference between each scanner used is that the Nessus output values include vulnerabilities over Nmap which has superior information on the services and ports of the machine. By using both, the optimal range of information can be achieved, however, this introduces the problem of overfitting which PCA has countered. For more information on PCA refer to dimensionality reduction section 1.6.4 in the introduction. It is possible to greatly modify the output of the two scanners by either using scripts with Nmap or custom plugins for Nessus. Due to the modularity of the application, these modifications to the scanners will not affect the parsers and therefore can be used safely. PCA is also implemented within the initialisation class

which can be found in Appendix A.3. Once PCA is complete, the still separated feature vectors are sent to the clusterer subclass and possibly the conditional merger. The following section will explain the conditional merger of which is represented by symbol in Figure 3.4.

3.2.3 Small Combined Cluster

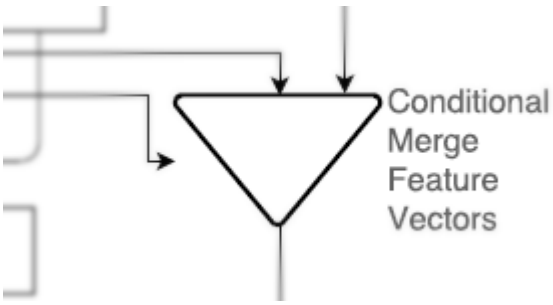


Figure 3.4: Conditional merge vectors process module from the infrastructure diagram at Figure 3.1

The conditional process of merging the feature vectors, referred to by the symbol in Figure 3.4, is executed at this stage, condition dependant on whether dual input mode has been selected when initialising the applications parameters. However, this path must then pause until the main cluster subclass (as represented by the symbol in Figure 3.6.) has completed in order to use its clustering outputs. Once this has occurred, each clustering will be duplicated and the large clusters with greater than three IP addresses will be removed from the clustering. This value can be changed based on the network size however the value of three was found to be optimal for networks of size 10 to 1000 from the algorithm tuning stage (this number can be configured for the users needs by modifying a single variable maxaddresses within the display class highlighted in Figure 3.5 below). The result of each is combined then re-clustered and the remaining IPs passed through a simpler file parser (compared to that of the ones previously used) to retrieve the information in an un-vectorised text format. This is done to display individual machine information for the end user within the graphing interface. The clustering of these small cluster IP addresses uses the gap statistic (or Elbow method if preferred by user) algorithm to define the numbers of clustered required as appose to user input as these IPs depend on the cluster subclass algorithm calculated clustering. The main cluster subclass mentioned is explained in

the following section 3.2.4. Gap statistic and Elbow method formulas can be found in the mathematical model at section 3.3.

```
142
143     def remove_large_clusters():
144
145         #originals needed for graph plotting t
146         nessusArray = clusterX
147         nmapArray = NclusterX
148         combined = np.array([])
149         maxaddresses = 3
150         #use biggest array by amount of cluste
151         #loop through largest cluster array, i
152         iterations=0
153         for index in range(len(nessusArray[:,
154             iplistnessus = nessusArray[(index-
155                 #delete largest clusters, clus
```

Figure 3.5: Location of the variable which defines the number of maximum IP addresses a cluster is allowed to have for the vulnerability analysis process. The variable is defined within the display.py classs remove large clusters function on line 149

3.2.4 Clustering Algorithm

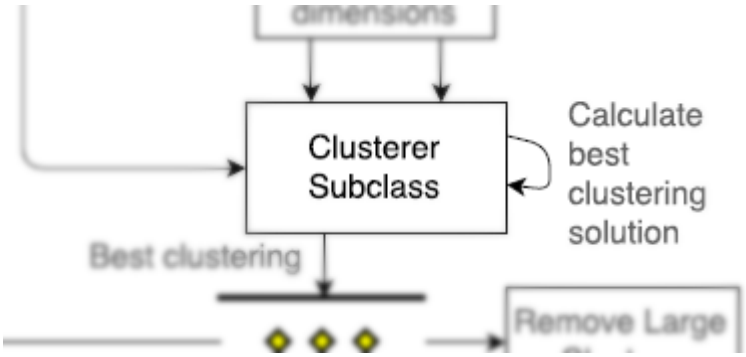


Figure 3.6: Representation of clusterer subclass from infrastructure diagram Figure 3.1 spotlighted

The Clusterer subclass represented in the infrastructure diagram by the module in Figure 3.6, is where the majority of the calculations are done within the application. This includes the three different modes; manual, assisted and automatic. The raw python code for this module can be found in Appendix A.5. Each mode can be used with a variety of clustering algorithms such as K-means, DBscan and agglomerative

clustering. When using dual input mode these functions will execute twice Nessus starting first. A brief summary of each mode is given in the following paragraphs whilst an in-depth algorithm review can be found in section 3.3:

Manual: The user supplies all required information to do the clustering. This includes the clustering algorithm and its hyper parameters. If no cluster count K is provided and K-means was selected, the gap statistic method will be used to calculate the optimal cluster count. An implementation of the elbow method is also included in the application and can be used instead if the gap statistic if the user desires.

Assisted: The user assists the algorithm by suggesting that some samples should or should not be clustered together. This is a much slower process than the other modes but provides the user with complete control over the clustering decisions. The user must specify the clustering algorithm to use from the selection mentioned above when using this mode.

Automatic: Multiple clustering strategies and parameters are used in an attempt to get the best clustering. Unfortunately, this function can only use the K-means clustering algorithm whilst running in dual input mode, due to limitations of the sklearn librarys centroid functions and the time constraints limiting this development as future work. This function goes through each possible iteration of each clustering algorithm up to a maximum cluster value equalling the number of labels in the given dataset. For e.g. in a dataset with ten IP addresses, the maximum allowed clusters is also then. There are multiple selections the user can decide on in terms of which clustering the application will select. This is calculated via a sorting function in the clustering subclass where the user can select between the average silhouette* value, the minimum silhouette* value, the average distance between IP addresses (Euclidian distance by default), the number of clusters in the clustering or the minimum number of common shared features a cluster has in its clustering. These options can also be joined together to make multiple filters. By default, the application uses both the minimum common features and number of overall clusters as filters allowing it to find the clustering with the least number of clusters requiring at least one shared feature. This option is changed near the end of the cluster subclass, changed by uncommenting one line per filter. Each clustering iteration in this loop will calculate the shared positive, negative features, the individual silhouette values and the mean distance to give an overall/per cluster score. Once the iterations are complete for each clustering algorithm the scores and details are used to select one of the clusterings based on the filters and then it is displayed. When using dual input mode, the vulnerability detection on the small clusters explained above in section 3.4 is continued,

due to it requiring these clusters as input. An example of the full text output and graphing interface for the applications automatic mode can be found at Appendix B.1.

* The silhouette value is used to validate the consistency of clusters in a clustering by indicating whether there are too many or too few clusters. It provides a value of negative one to positive one indicating how well each IP fits in the cluster. A high value close to one indicates a good clustering by showing that the IP address is very similar to the other IP addresses in its cluster, whilst being very different to the IP addresses of the neighbouring clusters. Therefore, having a low or negative silhouette value indicates the wrong number of clusters used. The silhouette value has been calculated using the Euclidian distance (described in section 1.6.2) within the validation class.

3.2.5 Covariance and Distance Matrices

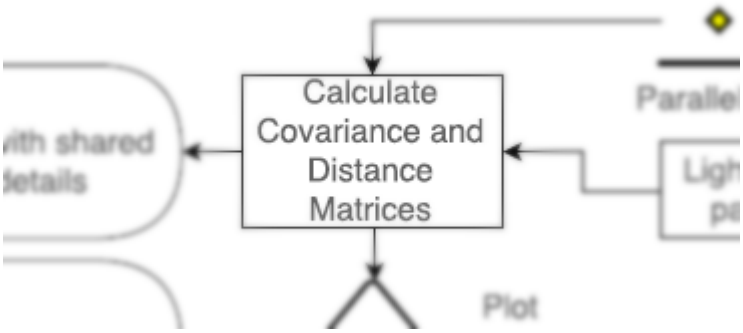


Figure 3.7: Representation of covariance and distance matrices function from infrastructure diagram Figure 3.1 spotlighted

Covariance and distance matrices are calculated just before displaying the graphing interface. The function symbol is shown in Figure 3.7 above. The output from these functions can be found at the end of the text output, an example is shown in Appendix B.1.1. This is shown for the purpose of giving a technical user greater knowledge of the data-set used which can be interpreted and reported to the client in a security assessment. Specifically, the covariance matrix shows how related the centroids for each clustering are to each other, in other words, their similarity. Covariance will highlight any linear relationships that might be apparent within the dataset with the values ranging from negative one to positive one, high values indicating a positive

linear relationship. The Distance matrix shows the literal Euclidian distance between each clustering centroid but by design that also correlates to showing a numericized value defining how different the clusters are from each other. These calculations are done for each clustering result in the current mode (For e.g. three times if using dual mode), however, they are only shown in verbose level one and above modes.

3.2.6 Display and Output Modules

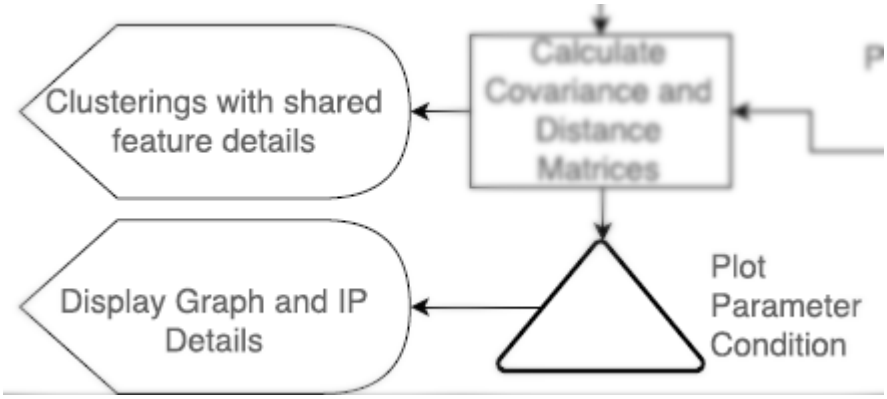


Figure 3.8: Representation of the display modules from infrastructure diagram Figure 3.1 with irrelevant modules blurred out

The application will display the main output at the end of its execution as shown in the infrastructure diagram with the symbol from Figure 3.8. The application will show a running output as the calculations and cluster iterations are performed, providing that the verbosity level is set to one or higher. The verbosity parameter will only effect the text output and not the graphing interface of the application which will remain identical, an example of full text output using dual mode can be found at B.1.1. Without verbose output selected the text output will only show the cluster details (for both formats if using dual mode). In order for the application to display the graphing interface on clustering class completion the application must be run with the parameter -p as mentioned in the usage descriptor found in A.2. The graphs displayed within the graphing GUI are fully manipulatable with zoom, pan and save as functions, however, the graphing interface will differ depending on which mode the application is using.

dual mode graphing interface will display five graphs and a small text segment underneath the bottom graph. The five segments are arranged in a grid of 4 with a large graph beneath the grid spanning both columns as the 5th. The first row of two graphs

displays the Nmap output with the right graph solely displaying the centroids and the left displaying the entire Nmap clustering with IP addresses. The second row displays the Nessus graphs in the same configuration as Nmap above. The 3rd row graph displays the IP addresses from the clusters with less than 3 IP addresses (this value is the default and can be changed by following the instruction described in section 3.4 and Figure 3.5). The reason for why this is done has been explained within the application brief at section 3.1.1 and the method used explained within section 3.4. The text segment beneath the 3rd row graph displays the detected operating system information and the probability of that detection being true. This is shown for each IP address in the small cluster IP addresses graph just above the text segment. An example of the dual mode graphing interface can be found in Figure B.1.2 and Figure B.4.

Standard mode graphing interfaces are similar to each other as they contain the same two graph split with just different information. The graphs are similar to that of dual mode without the small combined cluster graph of which the left graph includes the full clustering with IP addresses and the right graph solely contains the centroids of those clusters. Two examples of this mode can be found in the Appendix as Figure B.2 for Nessus clustering and Figure B.3 for Nmap clustering, along with the commands used to achieve them.

3.3 Mathematical Model and Algorithms

The following information consists of mathematical representations for the modules in the applications primary class architecture, as shown in figure 3.1. Parse XML files into matrices V for each input file, consisting of two dimensions m and n respectively. The matrices are then vectorised from JSON strings outputted from the parser classes into numerical float representations. These matrices' data is then normalized using the following formula denoted in the journal of Normalization: A Preprocessing Stage, Patro & Sahu (2015).: Where Vn is the nth dimension of matrix V and contains the features for each label Vm.

$V_n = \frac{(V_n - \min\{V_n\})}{(\max\{V_n\} - \min\{V_n\})} \left(new_{\max\{V_n\}} - new_{\min\{V_n\}}\right) + new\_\min\{V_n\}$

This application will normalise the data to the range of 0 to 1 in which the formula is adjusted to the following:

$V_n = \frac{(V_n - \min\{V_n\})}{(\max\{V_n\} - \min\{V_n\})}$

The Vn dimension of matrix V of each input file is then passed through a principle component analysis formula for dimensionality reduction to two dimensions. This is due to the current number of dimension layers directly correlating to a feature of which there is an average of three hundred per IP address (or Vm). This is calculated by using the SKlearn PCA decomposition library which uses the probabilistic model of PCA defined in the journal of Probabilistic Principle Component Analysis, Tipping & Bishop (1999). This algorithm is too complex to be included within the scope of this thesis and therefore is not described here. The result of this PCA algorithm returns a new matrix with two dimensions, which is then used in the K-means clustering algorithm as input with the IP address labels (or Vm).

The standard Lloyds algorithm for K-means clustering was then used against the PCA output vectors which can be described as the following five stages. This is carried out for each clustering iteration in the cluster class depending on its current mode.

1. Clusters the Vectors inputs into K number of clusters where K is either assigned manually, iteratively or calculated using the Gap statistic and Elbow values explained below. This depends on the current application mode and user parameters at run time.

2. Select K number of points at random locations to act as cluster centroids.

3. Assign objects to their closest plotted centroid per the selected distance function (Euclidian as default).

4. Calculate the new centroid for each cluster in this new clustering.

5. Repeat steps 2, 3 and 4 until the same points are assigned to each cluster in consecutive rounds and no changes are present in the clustering. In the case of automatic mode, the K value will increment up to a maximum equalling the number of IP addresses or labels in Vm.

However, the user can also select the use of DBscan or Agglomerative clustering algorithms instead of K-means for any mode apart from dual input. However, due to the limited usage they provide for the applications design goals, will not be described in this thesis.

Each clustering result of this algorithm has several calculations carried out upon it in order to validate and provide scores to the clustering. These calculations are denoted in section 3.6. The primary function used to score these clusterings is a silhouette value which has been defined in the journal, Silhouettes: A graphical aid to the interpretation and validation of cluster analysis, Rousseeuw (1987). A full description of what this value provides for the user is included at the end of section 3.6. The formulas used to calculate the silhouette value are taken from the previously mentioned journal and are described below.

For each vector i, *a(i)* would be the average dissimilarity of vector i with all other vectors within the same cluster. *b(i)* is the lowest average dissimilarity of vector i to any other cluster of which vector i is not a member of. The cluster with the lowest value for *b(i)* is the closest neighbouring cluster and therefore the most similar. This defines the following formula for silhouette value *s(i)*. $s\left(i\right) = \frac{b(i)-a(i)}{\max\{a(i),b(i)\}}$

The resulting clusterings are sorted based on the filters mentioned in 3.6 and the best clustering is chosen. This best clustering is passed through into the small combined cluster algorithm described in section 3.4 as well as the display class in 3.8.

When the application uses the K-means clustering algorithm and a value for K is not defined, the application will attempt to calculate the optimal value for K via either the Gap Statistic or Elbow value methods depending on the users preferences. This is changed within the optimal k means class with the default value being the Gap Statistic method. The python implementation of the Gap Statistic method can be found in the Appendix A.6 with the formulas used were originally developed by Stanford researchers from the journal, Estimating the number of clusters in a data set via the gap statistic, Tibshirani et al. (2001). The implementation of the researchers algorithm within this application can be described by the following points.

1. Cluster the data via K-means from the range of k=1 to k=max and calculate the variance quantity variable *Wk* (primarily used for elbow method).

2. Generate the reference data sets B and cluster them with the same k values as previous step.

3. Calculate the gap statistic value with the following formula: $Gap\,(k) = \left(\frac{1}{B}\right) \sum_{b=1}^{B} logW_{kb}^{*} - logW_{k}$.

4. Calculate the standard deviation using: $sd\,(k) = \left[\left(\frac{1}{B}\right) \sum_{b} (logW_{kb}^{*} - \left(\left(\frac{1}{B}\right) \sum_{b} logW_{kb}^{*}\right)^{2}\right]^{1/2}$

5. Then define $s_k = \sqrt{1 + \frac{1}{B}} sd\,(k)$ which allows the optimal number of K to be the smallest k such that $Gap\,(k) \geq Gap\,(k+1) - s_{k+1}$.

The Covariance and distance matrices displayed prior to graphing output are calculated using the following methods. More information on these values can be found in 3.2.5. Covariance for a given centroid vector dimensions x and y is calculated using the formula: $cov\,(x, y) = \frac{(\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}))}{n-1}$. This is calculated for each centroid in the chosen clustering and displayed in a matrix. The distance matrix displays the Euclidian distance between the x and y values for each centroid in the clustering.

3.4 Application Testing

Unfortunately due to the sensitive nature of the data required by the application, there are no datasets to be found publicly available online. Therefore, in order to design a proof of concept test procedure a dataset must be created using a significant network size to simulate the proportions of a real organizational network. The size requirements of this dataset limit the use of virtualized machine networks due the raw processing power required for a network of that size, hence, the permission was granted to allow for the scanning of a private network laboratory in the University of Abertay. The laboratory consisted of approximately 50 online devices at the time of scanning. The raw scan output files for Nmap and Nessus can be found within the thesis artefacts, hacklab analysis folder with the file names hacklab_new.xml and hacklab_new.nessus respectfully.

The creation of this dataset required installing the Nmap and Nessus scanner tools on a separate device plugged into the network for this sole purpose. The device used to enter the network and execute the scans was a Dell latitude E6410 laptop running the latest rolling release kernel of Arch Linux. This setup was used specifically to replicate a similar scenario to that of what an industry professional might encounter. The latest stable releases of the required software were used. At the time of writing this thesis (April 2017) those reference numbers were as follows:

- Nmap 7.40
- Nessus 6.10.5
- Python 2.7.13
- Python Libraries
  - backports-abc==0.4
  - bokeh==0.12.1
  - certifi==2016.8.8
  - cycler==0.10.0
  - futures==3.0.5
  - Jinja2==2.8
  - MarkupSafe==0.23
  - matplotlib==1.5.1
  - numpy==1.11.1
  - pyparsing==2.1.8
  - python-dateutil==2.5.3
  - pytz==2016.6.1
  - PyYAML==3.11
  - requests==2.11.0
  - scikit-learn==0.17.1
  - scipy==0.18.0
  - singledispatch==3.4.0.3
  - six==1.10.0
  - tornado==4.4.1
  - tabulate==0.7.7

The Nmap scan was created using the following command:

```
nmap -A -O -oX hacklab.xml 10.0.0.0/24
```

The -Ox flag in this command exports the scan as an XML which is the only export format the clustering application currently supports.
The Nessus scan was created by using the default scan policy in the advanced scan mode with the same scope as nmap, 10.0.0.0/24. This was then exported as the latest version of the proprietary Nessus XML format from the scan overview page.

Once both scans have completed the next step is to remove the testing device used from the XML files manually due to each scanner retrieving greatly differing results from the testing machine compared to other machines on the network. This step is crucial and ignoring it will manipulate the final result of the clustering algorithm rendering it inaccurate. The artefact files have had this step done prior to submission.

The scan files have then been passed through the clustering application using the following commands for each input type in automatic mode:

Nmap

```
cluster.py -s automatic -vv -p "../hacklab analyses/hacklab_new.xml"
```

Nessus

```
cluster.py -s automatic -vv -p -N "../hacklab analyses/hacklab_new.nessus"
```

Dual Input

```
cluster.py -s automatic -vv -p -t -N -tp "../hacklab
    analyses/hacklab_new.xml" "../hacklab analyses/hacklab_new.nessus"
```

More information and the results for this proof of concept test can be found in Appendix section B.2. These results will be validated within the thesis conclusions.

Chapter 4

Discussions and Results

Chapter 5

Conclusions

Appendix A

Code

A.1 Requirements python file

```
backports-abc==0.4
bokeh==0.12.1
certifi==2016.8.8
cycler==0.10.0
futures==3.0.5
Jinja2==2.8
MarkupSafe==0.23
matplotlib==1.5.1
numpy==1.11.1
pyparsing==2.1.8
python-dateutil==2.5.3
pytz==2016.6.1
PyYAML==3.11
requests==2.11.0
scikit-learn==0.17.1
scipy==0.18.0
singledispatch==3.4.0.3
six==1.10.0
tornado==4.4.1
tabulate==0.7.7
```

A.2 Application Usage Parameters

The following includes the usage and parameter definitions for the application.

```
usage: cluster.py [-h] [-s {manual,automatic,assisted}]
                  [-c {kmeans,dbscan,agglomerative}]
                  [--metric {euclidean,cosine,jaccard}] [-N] [-n N_CLUSTERS]
                  [-e EPSILON] [-m MIN_SAMPLES] [-cent] [-t] [-tp twinpath]
```

```
                [-p] [-v]
                path [path ...]
Cluster NMap/Nessus Output
positional arguments:
  path                Paths to files or directories to scan
optional arguments:
  -h, --help          show this help message and exit
  -s {manual,automatic,assisted}, --strategy {manual,automatic,assisted}
  -c {kmeans,dbscan,agglomerative}, --method {kmeans,dbscan,agglomerative}
  --metric {euclidean,cosine,jaccard}
  -N, --nessus        use .nessus file input
  -n N_CLUSTERS, --n_clusters N_CLUSTERS
                      Number of kmeans clusters to aim for
  -e EPSILON, --epsilon EPSILON
                      DBSCAN Epsilon
  -m MIN_SAMPLES, --min_samples MIN_SAMPLES
                      DBSCAN Minimum Samples
  -cent, --centroids  plot only centroids graph, requires the use of "-p"
  -t, --twin          use both input formats to calculate vulnerable single
                      clusters, use with -tp and -N
  -tp twinpath, --twinpath twinpath
                      path to nmap xml if using twin clustering
  -p, --plot          Plot clusters on 2D plane
  -v, --verbosity     increase output verbosity
```

A.3 Initialization Class

The following contains the raw code for the primary initializer class of file 'cluster.py' in the application parent folder. It is used to start the application using parameters from the usage, in Appendix A.2. The code has been thuroughly commented with the intention of being modified by a potential tester for individual operational requirements. This is further enforced by the application being entirely modular with each module able to be modified without harming the others. The cluster subclass has been removed from this code and placed in A.5 in order to promote the legibility of this appendix thus must not be executed direction without concatination with the afore mentioned subclass first.

```
import logging

from sklearn.preprocessing import normalize

from clusterer_parts.optimal_k_k_means import optimalK
```

```
from clusterer_parts.analysis import get_common_features_from_cluster,
    get_common_feature_stats
from clusterer_parts.clustering import cluster_with_dbscan,
    cluster_with_kmeans, precompute_distances, \
    cluster_with_agglomerative, cluster_interactive, get_centroids,
        cluster_single_kmeans, get_k
from clusterer_parts.display import print_cluster_details,
    generate_dot_graph_for_gephi, create_plot, \
    create_plot_centroids, create_plot_only_centroids, twin,
        remove_large_clusters
from clusterer_parts.optimizing import sort_items_by_multiple_keys
from clusterer_parts.reduction import pca
from clusterer_parts.validation import validate_clusters,
    get_average_distance_per_cluster
from clusterer_parts.vectorize import vectorize
import numpy as np
from tabulate import tabulate
firstpass = True

if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description=u'Cluster NMap/Nessus
        Output')

    parser.add_argument('path', metavar='path', type=str, nargs='+',
        default=None,
                        help="Paths to files or directories to scan")

    parser.add_argument('-s', '--strategy', default="automatic",
        choices=["manual", "automatic", "assisted"])
    parser.add_argument('-c', '--method', default="kmeans",
        choices=["kmeans", "dbscan", "agglomerative"])
    parser.add_argument('--metric', default="euclidean",
        choices=["euclidean", "cosine", "jaccard"])
    parser.add_argument('-N', '--nessus', default="false", required=False,
        action='store_true',
                        help='use .nessus file input')

    parser.add_argument('-n', '--n_clusters', type=int, default=2,
        help='Number of kmeans clusters to aim for')
    parser.add_argument('-e', '--epsilon', type=float, default=0.5,
        help='DBSCAN Epsilon')
    parser.add_argument('-m', '--min_samples', type=int, default=5,
        help='DBSCAN Minimum Samples')
    parser.add_argument('-cent', '--centroids', default=False,
        required=False, action='store_true',
```

```
                    help='plot only centroids graph, requires the use of
                        "-p"')
parser.add_argument('-t', '--twin', default=False, required=False,
    action='store_true',
                    help='use both input formats to calculate vulnerable
                        single clusters, use with -tp and -N')
parser.add_argument('-tp', '--twinpath', metavar='twinpath', type=str,
    required=False,
                    help='path to nmap xml if using twin clustering')

parser.add_argument('-p', '--plot', default=False, required=False,
    action='store_true',
                    help='Plot clusters on 2D plane')

parser.add_argument("-v", "--verbosity", action="count",
    help="increase output verbosity")
args = parser.parse_args()

logging.basicConfig(format='\%(asctime)s \%(process)s \%(module)s
    \%(funcName)s \%(levelname)-8s :\%(message)s',
                    datefmt='\%m-\%d \%H:\%M')

if args.verbosity == 1:
    logging.getLogger().setLevel(logging.INFO)
elif args.verbosity > 1:
    logging.getLogger().setLevel(logging.DEBUG)

if (args.twin == False):

    # Vectorize our input
    logging.info("Vectorizing Stage")
    vector_names, vectors, vectorizer = vectorize(args.path,
        args.nessus)
    logging.debug("Loaded {0} vectors with {1}
        features".format(len(vector_names), vectors.shape[1]))
    logging.info("Vectorizing complete")

    # normalise vectors first before passing them through PCA. PCA uses
        2 dimensions
    logging.info("Normalising the vectors")
    normalized_vectors = normalize(vectors)
    logging.info("Reducing vectors to two dimensions with PCA")
    reduced_vectors = pca(normalized_vectors)
    logging.debug(
        "reduced to {0} vectors with {1}
            dimensions".format((reduced_vectors.shape[0]),
            reduced_vectors.shape[1]))
```

```
# Cluster the vectors
logging.info("Clustering")
labels = cluster(vector_names, vectors, reduced_vectors,
    normalized_vectors, vectorizer, args.strategy,
                args.method, args.n_clusters, args.epsilon,
                    args.min_samples, args.metric)
logging.info("Clustering Complete")
# Test cluster validity
overall_score, per_cluster_score = validate_clusters(vectors,
    labels)

# Analysis relevant to the person reading results
universal_positive_features, universal_negative_features,
    shared_features = get_common_features_from_cluster(
    vectors, labels, vectorizer)

# logging.debug("Shared features: {0}".format(shared_features))

# Reduce results and relevant information to per cluster data
cluster_details = {}
for cluster_id in per_cluster_score.keys():
    cluster_details[cluster_id] = {
        "silhouette": per_cluster_score[cluster_id],
        "shared_positive_features":
            shared_features[cluster_id]['positive'],
        # "shared_negative_features":
            shared_features[cluster_id]['negative'],
        "ips": [vector_names[x] for x in xrange(len(vector_names))
            if labels[x] == cluster_id]
    }
print "Note: shared features does not retain keys from XML and
    therefore wont always be human readable."
print_cluster_details(cluster_details, shared_features)

if args.plot:
    # only kmeans centroids for now
    if no_clusters.startswith("kmeans") :
        logging.debug("Getting centroids using reduced vectors:")
        # global centroidskmeans
        # take just cluster number from result string
        n_clusters = no_clusters.split("=", 1)[1]
        n_clusters = int(n_clusters.rsplit(')', 1)[0])
        logging.debug("nclusters: " + str(n_clusters))

        centroidskmeans = get_centroids(reduced_vectors, n_clusters)
        logging.debug("attempting to plot the following centroids:\n
```

```
            " + str(centroidskmeans))

        # covariance
        x = centroidskmeans[:, 0]
        y = centroidskmeans[:, 1]
        X = np.vstack((x, y))
        cov = np.cov(X)
        logging.info("Centroids Covariance Matrix:\n
            {0}".format(cov))

        # print similarity distance between centroids
        matrix = precompute_distances(centroidskmeans,
            metric=args.metric)
        matrixTable = tabulate(matrix)
        logging.info(
            "distance matrix between centroids using metric: {0}
                :\n{1}".format(args.metric, matrixTable))

        if args.centroids:
            create_plot_only_centroids(reduced_vectors, labels,
                vector_names, centroidskmeans, n_clusters)
        else:
            create_plot_centroids(reduced_vectors, labels,
                vector_names, centroidskmeans, n_clusters,
                                cluster_details)

    # manually selected kmeans though arguments
    elif args.method == "kmeans" and args.strategy != "automatic":
        if get_k()>0:
            centroidskmeans = get_centroids(reduced_vectors, get_k())
        else:
            centroidskmeans = get_centroids(reduced_vectors,
                args.n_clusters)

        logging.debug("attempting to plot the following centroids:
            \n" + str(centroidskmeans))

        # covariance
        x = centroidskmeans[:, 0]
        y = centroidskmeans[:, 1]
        X = np.vstack((x, y))
        cov = np.cov(X)
        logging.info("Centroids Covariance Matrix:\n
            {0}".format(cov))

        # print similarity distance between centroids
        matrix = precompute_distances(centroidskmeans,
```

```
                metric=args.metric)
            matrixTable = tabulate(matrix)
            logging.info(
                "distance matrix between centroids using metric: {0}
                    :\n{1}".format(args.metric, matrixTable))

            if args.centroids:
                if get_k() > 0:
                    create_plot_only_centroids(reduced_vectors, labels,
                        vector_names, centroidskmeans, get_k())
                else:
                    create_plot_only_centroids(reduced_vectors, labels,
                        vector_names, centroidskmeans, args.n_clusters)
            else:
                if get_k() > 0:
                    create_plot_centroids(reduced_vectors, labels,
                        vector_names, centroidskmeans, get_k(),
                                    cluster_details)
                else:
                    create_plot_centroids(reduced_vectors, labels,
                        vector_names, centroidskmeans, args.n_clusters,
                                    cluster_details)
        else:
            logging.debug("plotting standard graph")
            create_plot(reduced_vectors, labels, vector_names)
    # Write DOT diagram out to cluster.dot, designed for input into
        Gephi (https://gephi.org/)
    with open("cluster.dot", "w") as f:
        f.write(
            generate_dot_graph_for_gephi(precompute_distances(vectors,
                metric=args.metric), vector_names, labels))

elif args.twin == True and args.strategy == "automatic":

    logging.debug("twin flag enabled")
    logging.debug("tp: {0} , path: {1}".format(args.twinpath,
        args.path))

    # Vectorize our input for nessus
    logging.info("Vectorizing Stage for Nessus")
    Nvector_names, Nvectors, Nvectorizer = vectorize(args.path,
        args.nessus)
    logging.debug("Loaded {0} vectors with {1}
        features".format(len(Nvector_names), Nvectors.shape[1]))
    logging.info("Vectorizing complete\n")

    # Vectorize our input for nmap
```

```
logging.info("Vectorizing Stage for nmap")
twinpath = list()
twinpath.append(args.twinpath)
vector_names, vectors, vectorizer = vectorize(twinpath, False)
logging.debug("Loaded {0} vectors with {1}
    features".format(len(vector_names), vectors.shape[1]))
logging.info("Vectorizing complete\n")

# normalise vectors first before passing them through PCA. PCA uses
    2 dimensions
# nessus
logging.info("Normalising the nessus vectors")
Nnormalized_vectors = normalize(Nvectors)
logging.info("Reducing vectors to two dimensions with PCA")
Nreduced_vectors = pca(Nnormalized_vectors)
logging.debug(
    "reduced to {0} vectors with {1}
        dimensions".format((Nreduced_vectors.shape[0]),
        Nreduced_vectors.shape[1]))
logging.info("Normalising complete\n")

# normalise vectors first before passing them through PCA. PCA uses
    2 dimensions
# nmap
logging.info("Normalising the nmap vectors")
normalized_vectors = normalize(vectors)
logging.info("Reducing vectors to two dimensions with PCA")
reduced_vectors = pca(normalized_vectors)
logging.debug(
    "reduced to {0} vectors with {1}
        dimensions".format((reduced_vectors.shape[0]),
        reduced_vectors.shape[1]))
logging.info("Normalising complete\n")

# Cluster the vectors
logging.info("Clustering Nessus")
Nlabels = cluster(Nvector_names, Nvectors, Nreduced_vectors,
    Nnormalized_vectors, Nvectorizer, args.strategy,
                args.method, args.n_clusters, args.epsilon,
                    args.min_samples, args.metric)
logging.info("Clustering Complete\n\n")
# Test cluster validity
Noverall_score, Nper_cluster_score = validate_clusters(Nvectors,
    Nlabels)

# Cluster the vectors
logging.info("Clustering Nmap")
```

```
labels = cluster(vector_names, vectors, reduced_vectors,
    normalized_vectors, vectorizer, args.strategy,
                args.method, args.n_clusters, args.epsilon,
                    args.min_samples, args.metric)
logging.info("Clustering Complete\n\n")
# Test cluster validity
overall_score, per_cluster_score = validate_clusters(vectors,
    labels)

# Analysis relevant to the person reading results
# nessus
Nuniversal_positive_features, Nuniversal_negative_features,
    Nshared_features = get_common_features_from_cluster(
    Nvectors, Nlabels, Nvectorizer)

# Analysis relevant to the person reading results
# nmap
universal_positive_features, universal_negative_features,
    shared_features = get_common_features_from_cluster(
    vectors, labels, vectorizer)

# Reduce results and relevant information to per cluster data
# nessus
Ncluster_details = {}
for cluster_id in Nper_cluster_score.keys():
    Ncluster_details[cluster_id] = {
        "silhouette": Nper_cluster_score[cluster_id],
        "shared_positive_features":
            Nshared_features[cluster_id]['positive'],
        "ips": [Nvector_names[x] for x in xrange(len(Nvector_names))
            if Nlabels[x] == cluster_id]
    }
print "Note: shared features does not retain keys from XML and
    therefore wont always be human readable."
print "Printing Nessus cluster details\n"
print_cluster_details(Ncluster_details, Nshared_features)

print "\n\n"

# Reduce results and relevant information to per cluster data
cluster_details = {}
for cluster_id in per_cluster_score.keys():
    cluster_details[cluster_id] = {
        "silhouette": per_cluster_score[cluster_id],
        "shared_positive_features":
            shared_features[cluster_id]['positive'],
        # "shared_negative_features":
```

```
            shared_features[cluster_id]['negative'],
        "ips": [vector_names[x] for x in xrange(len(vector_names))
            if labels[x] == cluster_id]
    }
print "Printing Nmap cluster details\n"
print_cluster_details(cluster_details, shared_features)

if args.plot:
        # Nmap
        logging.debug("Getting centroids using reduced vectors for
            Nmap:")
        # take just cluster number from result string
        n_clusters = Nno_clusters.split("=", 1)[1]
        n_clusters = int(n_clusters.rsplit(')', 1)[0])
        logging.debug("nclusters: " + str(n_clusters))

        centroidskmeans = get_centroids(reduced_vectors, n_clusters)
        k = get_k()
        logging.debug("attempting to plot the following centroids:\n
            " + str(centroidskmeans) + "\n\n")

        # Nessus
        logging.debug("Getting centroids using reduced vectors for
            Nessus:")
        # take just cluster number from result string
        logging.debug("nclusters: " + str(Nno_clusters))

        Nn_clusters = no_clusters.split("=", 1)[1]
        Nn_clusters = int(Nn_clusters.rsplit(')', 1)[0])
        logging.debug("nclusters: " + str(Nn_clusters))

        Ncentroidskmeans = get_centroids(Nreduced_vectors,
            Nn_clusters)
        logging.debug("attempting to plot the following centroids:\n
            " + str(Ncentroidskmeans) + "\n\n")
        Nk = get_k()

        # covariance for Nmap
        x = centroidskmeans[:, 0]
        y = centroidskmeans[:, 1]
        X = np.vstack((x, y))
        cov = np.cov(X)
        logging.info("Nmap Centroids Covariance Matrix:\n
            {0}".format(cov))

        # covariance for Nessus
        Nx = Ncentroidskmeans[:, 0]
```

```
Ny = Ncentroidskmeans[:, 1]
NX = np.vstack((Nx, Ny))
Ncov = np.cov(NX)
logging.info("Nessus Centroids Covariance Matrix:\n
    {0}".format(Ncov))

# print similarity distance between centroids
# Nessus
matrix = precompute_distances(centroidskmeans,
    metric=args.metric)
matrixTable = tabulate(matrix)
logging.info(
    "distance matrix between centroids using metric for
        Nmap: {0} :\n{1}".format(args.metric, matrixTable))

# print similarity distance between centroids
# Nmap
Nmatrix = precompute_distances(Ncentroidskmeans,
    metric=args.metric)
NmatrixTable = tabulate(Nmatrix)
logging.info(
    "distance matrix between centroids using metric for
        Nessus: {0} :\n{1}".format(args.metric,
        NmatrixTable))

small_ips = remove_large_clusters()

logging.info("IP's from clusters with less than 3 IP's:\n
    {0}".format((small_ips)))


#creates large array with 2nd dimension as large enough to
    hold both feature vectors
nesmap = np.zeros((len(small_ips),
    (Nvectors.shape[1]+vectors.shape[1])))


#for each single ip
for index in range(len(small_ips)):
    # for each ip in vectors
    features = 0
    for index2 in range(vectors.shape[0]):
        #if ip is equal to vector ip
        #logging.debug("if {0} = {1}
            ".format(small_ips[index], vector_names[index2]))
        if small_ips[index] == vector_names[index2]:
            #logging.debug("ip is equal to vector ip")
```

```
            #for every one of this vectors features
            for index3 in range(vectors.shape[1]):
                #assign its features to single ip vector
                nesmap[index,features] = vectors[index2,
                    index3]
                features +=1
            break

    for index2 in range(Nvectors.shape[0]):
        #logging.debug("if {0} = {1}
            ".format(small_ips[index], Nvector_names[index2]))
        if small_ips[index] == Nvector_names[index2]:
            #logging.debug("ip is equal to vector ip")
            for index3 in range(Nvectors.shape[1]):
                #append nessus features onto nmap features
                nesmap[index,features] = Nvectors[index2,
                    index3]
                features += 1
            break

logging.debug("Loaded {0} vectors with {1}
    features".format(nesmap.shape[0], nesmap.shape[1]))
small_normalized_vectors = normalize(nesmap)
logging.info("Normalizing input and reducing vectors to two
    dimensions with PCA")
final = pca(small_normalized_vectors)

logging.info("Resulting single IP vectors:\n
    {0}".format(final))

Smatrix = precompute_distances(final, metric=args.metric)
SmatrixTable = tabulate(Smatrix)
logging.info(
    "distance matrix between centroids of small combined
        clusters: {0} :\n{1}".format(args.metric,
        SmatrixTable))
clusterz = cluster_single_kmeans(final, 2)

logging.info("Writing recommended attack IP's to targets.txt
    for exploitation\n {0}")
f = open('targets.txt', 'w')
for index in range(len(small_ips)):
    f.write('{0}\n'.format(small_ips[index])) # python will
        convert \n to os.linesep
f.close() # you can omit in most cases as the destructor
    will call it
```

```
            twin(reduced_vectors, labels, vector_names, centroidskmeans,
                n_clusters, cluster_details, Nreduced_vectors, Nlabels,
                Nvector_names, Ncentroidskmeans, Nn_clusters,
                Ncluster_details, small_ips, final, clusterz, twinpath)

    else: print "not yet implemented #todo"
```

A.4 vectorization class - vectorize.py

The following contains the raw code for the vectorize.py class. This class is called by the main cluster.py class at appendix A.3 in order to extract the information from the scan files as well as vectorize that information to be returned to the primary class.

```
import logging
from parsing import parsers
from parse_nessus import Nparsers
from single_ip_parsing_nmap import single_ip_parsers
import numpy as np


class Vectorizer:
    """
    This class handles the vectorizing of input
    It additionally stores all pseudo vectors until we are ready for the
        finished vectors
    """

    def __init__(self):
        self.tokenized_strings = []
        self.pseudo_vectors = {}

    def add_string_to_ip(self, ip, string):
        if ip not in self.pseudo_vectors:
            self.pseudo_vectors[ip] = []

        if string not in self.tokenized_strings:
            self.tokenized_strings.append(string)

        s_id = self.tokenized_strings.index(string)
        self.pseudo_vectors[ip].append(s_id)

    def parse_input(self, input_string, n):
    ##n boolean value refers to nessus input only
        if (n==True):
            for parser in Nparsers:
```

```
                if parser.can_parse_input(input_string):
                    results = parser.parse_input(input_string)
                    for key in results.keys():
                        for s in results[key]:
                            self.add_string_to_ip(key, s)
        else:
            for parser in parsers:
                if parser.can_parse_input(input_string):
                    results = parser.parse_input(input_string)
                    for key in results.keys():
                        for s in results[key]:
                            self.add_string_to_ip(key, s)

    def output_vectors(self):
        vector_names = []
        vectors = np.zeros((len(self.pseudo_vectors.keys()),
            len(self.tokenized_strings)), dtype=np.float)

        for ip_index, ip in enumerate(self.pseudo_vectors.keys()):
            vector_names.append(ip)
            for s_index in self.pseudo_vectors[ip]:
                # Just set it to one, we want to ignore any case we see a
                    value more than once
                vectors[ip_index, s_index] = 1

        return vector_names, vectors


def vectorize(files_to_vectorize, n):
    vectorizer = Vectorizer()
    for file_path in files_to_vectorize:
        with open(file_path, "r") as f:
            vectorizer.parse_input(f.read(), n)

    vector_names, vectors = vectorizer.output_vectors()

    return vector_names, vectors, vectorizer

def parse_single_ips(files_to_vectorize, ips):
    for file_path in files_to_vectorize:
        with open(file_path, "r") as f:
            for parser in single_ip_parsers:
                #logging.debug("vectorisor selecting single ips")
                results = parser.parse_input(f.read(), ips)
                return results
```

A.5 Clustering Algorithm subclass

The following code consists of the clustering algorithm used for all the modes within the application. It uses the provided parameters to decide between these modes as well as calls several other subclasses for code modularity.

```
def cluster(
        vector_names,
        vectors,
        reduced_vectors,
        normalized_vectors,
        vectorizer,
        strategy="automatic",
        cluster_method="kmeans",
        n_clusters=2,
        epsilon=0.5,
        min_samples=5,
        metric="euclidean",
):
    """
    Clustering options:

    Manual:
     The user supplies all required information to do the clustering. This
         includes the clustering algorithm and
     hyper parameters,
     if no cluster count is provided the gap_statistic method will be used
         to calculate the optimal cluster count

    Assisted:
     The user assists the algorithm by suggesting that some samples should
         or should not be clustered together

    Automatic:
     The multiple clustering strategies and parameters are used in an
         attempt to get the best clusters

     finds the least amount of clusters with atleast one shared feature

     only uses gap statistic for small IP clusters
    """

    global centroidskmeans, centroidagglo, centroiddbs, no_clusters,
        Nno_clusters, Ncentroidskmeans
```

```
if strategy == "manual":
    no_clusters = ""
    if cluster_method == "kmeans":
        #centroidskmeans = get_centroids(reduced_vectors,
            n_clusters=n_clusters)
        #logging.debug("centroids for kmeans:
            {0}".format(centroidskmeans))
        k, gapdf = optimalK(vectors, nrefs=3,
            maxClusters=reduced_vectors.shape[0])
        return cluster_with_kmeans(reduced_vectors, n_clusters=k)

    elif cluster_method == "dbscan":
        return cluster_with_dbscan(reduced_vectors, epsilon=epsilon,
            min_samples=min_samples, metric=metric)

    elif cluster_method == "agglomerative":
        return cluster_with_agglomerative(reduced_vectors,
            n_clusters=n_clusters, metric=metric)

    else:
        # Unknown clustering method
        raise NotImplementedError()

elif strategy == "assisted":
    """
    To display a information about a vector to a user, you can use the
        following:
    display_vector_index_details(vector_index, vectors, vector_names,
        vectorizer)
    """

    return cluster_interactive(reduced_vectors, vectorizer, vectors,
        vector_names)
elif strategy == "automatic":
    results = []
    smallest_cluster_count = vectors.shape[0]
    # centroids works for only kmeans atm
    for cluster_method in [
        #todo add agglo and dbscan back in after they can return
            centroids.
        "kmeans" # ,
        # "agglomerative",
        # "dbscan",
    ]:
        if cluster_method == "kmeans":
            #this method is called X-means clustering
            logging.debug("Starting prospective KMeans clusterings")
```

```
move_to_next_method = False
# start at 2 clusters and end at smallest_cluster_count
for n_clusters in xrange(2, smallest_cluster_count):
    logging.debug("Trying
        {0}".format("kmeans(n_clusters={0})".format(n_clusters)))
    labels = cluster_with_kmeans(reduced_vectors,
        n_clusters=n_clusters)
    overall_score, per_cluster_score =
        validate_clusters(vectors, labels)
    mean_distance =
        get_average_distance_per_cluster(vectors, labels)[0]

    tsp, msp, msn = get_common_feature_stats(vectors,
        labels, vectorizer)

    # If any cluster has 0 shared features, we just ignore
        the result
    if msp <= tsp:
        logging.debug("Not all clusters are informative (a
            cluster has 0 shared features) ")
        continue
    if len(set(labels)) > smallest_cluster_count:
        move_to_next_method = True
        # logging.debug("len(set(labels)): {0} >
            smallest_cluster_count:
            {1}".format(len(set(labels)),
            smallest_cluster_count))
        break
    if len(set(labels)) < smallest_cluster_count:
        smallest_cluster_count = len(set(labels))
    #too verbose
    # logging.debug(repr((
    #         overall_score,
    #         min(per_cluster_score.values()),
    #         mean_distance,
    #         labels,
    #         len(set(labels)),
    #         tsp,
    #         msp,
    #         msn,
    #         "kmeans(n_clusters={0})".format(n_clusters)
    #     )))
    results.append(
        (
            overall_score,
            min(per_cluster_score.values()),
            mean_distance,
```

```
                    labels,
                    len(set(labels)),
                    tsp,
                    msp,
                    msn,
                    "kmeans(n_clusters={0})".format(n_clusters)
                )
            )
        if move_to_next_method:
            continue

    if cluster_method == "agglomerative":
        logging.debug("Starting prospective Agglomerative
            clusterings")
        move_to_next_method = False
        for n_clusters in xrange(2, smallest_cluster_count):
            logging.debug("Trying
                {0}".format("agglomerative(n_clusters={0})".format(n_clusters)))
            labels = cluster_with_agglomerative(reduced_vectors,
                n_clusters=n_clusters, metric=metric)
            overall_score, per_cluster_score =
                validate_clusters(vectors, labels)
            mean_distance =
                get_average_distance_per_cluster(vectors, labels)[0]

            tsp, msp, msn = get_common_feature_stats(vectors,
                labels, vectorizer)

            # If any cluster has 0 shared features, we just ignore
                the result
            if msp <= tsp:
                logging.debug("Not all clusters are informative (a
                    cluster has 0 shared features) ")
                continue
            if len(set(labels)) > smallest_cluster_count:
                move_to_next_method = True
                break
            if len(set(labels)) < smallest_cluster_count:
                smallest_cluster_count = len(set(labels))

            logging.debug(repr((
                overall_score,
                min(per_cluster_score.values()),
                mean_distance,
                labels,
                len(set(labels)),
                tsp,
```

```
                    msp,
                    msn,
                    "agglomerative(n_clusters={0})".format(n_clusters)
                )))
                results.append(
                    (
                        overall_score,
                        min(per_cluster_score.values()),
                        mean_distance,
                        labels,
                        len(set(labels)),
                        tsp,
                        msp,
                        msn,
                        "agglomerative(n_clusters={0})".format(n_clusters)
                    )
                )
            if move_to_next_method:
                continue

        if cluster_method == "dbscan":
            logging.debug("Starting prospective DBSCAN clusterings")
            distance_matrix = precompute_distances(vectors,
                metric=metric)
            min_distance =
                sorted(set(list(distance_matrix.flatten())))[1]
            max_distance =
                sorted(set(list(distance_matrix.flatten())))[-1]
            num_steps = 25.0
            step_size = float(max_distance - min_distance) /
                float(num_steps)
            epsilon = min_distance
            while True:
                logging.debug("Trying
                    {0}".format("dbscan(epsilon={0})".format(epsilon)))
                labels = cluster_with_dbscan(reduced_vectors,
                    epsilon=epsilon, min_samples=1,
                                                distances=distance_matrix)
                if len(set(labels)) == 1 and list(set(labels))[0] == 0:
                    break
                overall_score, per_cluster_score =
                    validate_clusters(vectors, labels)
                mean_distance =
                    get_average_distance_per_cluster(vectors, labels)[0]

                tsp, msp, msn = get_common_feature_stats(vectors,
                    labels, vectorizer)
```

```
            # If any cluster has 0 shared features, we just ignore
                the result
            if msp <= tsp:
                logging.debug("Not all clusters are informative (a
                    cluster has 0 shared features) ")
                epsilon += step_size
                continue

            logging.debug(repr((
                overall_score,
                min(per_cluster_score.values()),
                mean_distance,
                labels,
                len(set(labels)),
                tsp,
                msp,
                msn,
                "dbscan(epsilon={0})".format(epsilon)
            )))
            results.append(
                (
                    overall_score,
                    min(per_cluster_score.values()),
                    mean_distance,
                    labels,
                    len(set(labels)),
                    tsp,
                    msp,
                    msn,
                    "dbscan(epsilon={0})".format(epsilon)
                )
            )
            epsilon += step_size

    # Choose best clustering result based on the following attributes
    sorted_results = sort_items_by_multiple_keys(
        results,
        {
            # 0: True, # AVG Silhouette
            # 1: True, # Min Silhouette
            # 2: False, # Average distance
            4: False, # Number of clusters
            # 6: True,  # Min common features per cluster
        },
        {
            # 0: 1,
```

```
                # 1: 1,
                # 2: 1,
                4: 1,
                # 6: 1
            }
        )
        # logging.debug(sorted_results)
        best_result = results[sorted_results[0][0]]
        # logging.debug(best_result)

        best_method = best_result[-1]
        best_silhouette = best_result[0]
        best_labels = best_result[3]
        global firstpass
        if firstpass:
            no_clusters = best_result[-1]
            firstpass = False
        else:
            Nno_clusters = best_result[-1]

        # no_clusters = best_result[-1]

        logging.info("Best clustering method: {0} (adjusted silhouette ==
            {1})".format(best_method, best_silhouette))
        return best_labels

    else:
        # Unknown strategy
        raise NotImplementedError()
```

A.6 Gap Statistic Implementation

The following segment of codeis the defined function 'optimalK' which calculates the optimal value for K in K-means via the Gap Statistic method. This implementation is within the optimal_k_k_means.py class and is thoroughly commentated. The parameters and returned value are explained at the top of the function.

```
def optimalK(data, nrefs, maxClusters):
    """
    Calculates KMeans optimal K using Gap Statistic from
        http://web.stanford.edu/~hastie/Papers/gap.pdf
    Params:
        data: ndarry of shape (n_samples, n_features)
        nrefs: number of sample reference datasets to create
```

```
    maxClusters: Maximum number of clusters to test for
Returns: (gaps, optimalK)
"""
gaps = np.zeros((len(range(1, maxClusters)),))
resultsdf = pd.DataFrame({'clusterCount': [], 'gap': []})
for gap_index, k in enumerate(range(1, maxClusters)):

    # Holder for reference dispersion results
    refDisps = np.zeros(nrefs)

    # For n references, generate random sample and perform kmeans
        getting resulting dispersion of each loop
    for i in range(nrefs):
        # Create new random reference set
        randomReference = np.random.random_sample(size=data.shape)

        # Fit to it
        km = KMeans(k)
        km.fit(randomReference)

        refDisp = km.inertia_
        refDisps[i] = refDisp

    # Fit cluster to original data and create dispersion
    km = KMeans(k)
    km.fit(data)

    origDisp = km.inertia_

    # Calculate gap statistic
    gap = np.log(np.mean(refDisps)) - np.log(origDisp)

    # Assign this loop's gap statistic to gaps
    gaps[gap_index] = gap

    resultsdf = resultsdf.append({'clusterCount': k, 'gap': gap},
        ignore_index=True)

return (gaps.argmax() + 1, resultsdf) # Plus 1 because index of 0
    means 1 cluster is optimal, index 2 = 3 clusters are optimal
```

Appendix B

Example Application Output

B.1 Fictional Small Network Output

This output was generated by running the application in automatic, maximum verbosity, dual input with graph plotting mode. The command used to execute the application with this configuration is as follows:

```
cluster.py -s automatic -vv -p -t -N -tp ../cw.xml ../cw.nessus
```

The network in which the scans were taken place is an entirely fictional and uses four machines in the scope. This scope is described below.

Table B.1: Fictional Network Scope and Machine Descriptions

192.168.0.1	Windows 2008 server running Apache 2 web server with MySQL database. Hosting a DNS server. Domain Controller for UADTARGETNET domain running Lightweight Directory Access Protocol NETBIOS name: SERVER1
192.168.0.2	Windows 2008 server running empty web server and hosting a DNS server running Lightweight Directory Access Protocol NETBIOS name: SERVER2
192.168.0.10	Windows 7 Professional 7600 (Windows 7 Professional 6.1) NETBIOS name: CLIENT1
192.168.0.11	Windows 7 Professional 7600 (Windows 7 Professional 6.1) NETBIOS name: CLIENT2

B.1.1 Text Output

```
04-19 01:32 19624 cluster <module> DEBUG :twin flag enabled
04-19 01:32 19624 cluster <module> DEBUG :tp: ../cw.xml , path: ['../cw.nessus']
04-19 01:32 19624 cluster <module> INFO :Vectorizing Stage for Nessus
04-19 01:32 19624 parse_nessus parse_input INFO :Parsing Nessus XML * BETA *
no of IP's taken from nessus: 4
04-19 01:32 19624 parse_nessus parse_input INFO :Done Nessus parsing
04-19 01:32 19624 cluster <module> DEBUG :Loaded 4 vectors with 45 features
04-19 01:32 19624 cluster <module> INFO :Vectorizing complete

04-19 01:32 19624 cluster <module> INFO :Vectorizing Stage for nmap
no of IP's taken from NMAP: 4
04-19 01:32 19624 cluster <module> DEBUG :Loaded 4 vectors with 81 features
04-19 01:32 19624 cluster <module> INFO :Vectorizing complete

04-19 01:32 19624 cluster <module> INFO :Normalising the nessus vectors
04-19 01:32 19624 cluster <module> INFO :Reducing vectors to two dimensions with PCA
04-19 01:32 19624 cluster <module> DEBUG :reduced to 4 vectors with 2 dimensions
04-19 01:32 19624 cluster <module> INFO :Normalising complete

04-19 01:32 19624 cluster <module> INFO :Normalising the nmap vectors
04-19 01:32 19624 cluster <module> INFO :Reducing vectors to two dimensions with PCA
04-19 01:32 19624 cluster <module> DEBUG :reduced to 4 vectors with 2 dimensions
04-19 01:32 19624 cluster <module> INFO :Normalising complete

04-19 01:32 19624 cluster <module> INFO :Clustering Nessus
04-19 01:32 19624 cluster cluster DEBUG :Starting prospective KMeans clusterings
04-19 01:32 19624 cluster cluster DEBUG :Trying kmeans(n_clusters=2)
04-19 01:32 19624 cluster cluster DEBUG :Trying kmeans(n_clusters=3)
04-19 01:32 19624 cluster cluster INFO :Best clustering method: kmeans(n_clusters=2) (adjusted
    silhouette == 0.176059056707)
04-19 01:32 19624 cluster <module> INFO :Clustering Complete


04-19 01:32 19624 cluster <module> INFO :Clustering Nmap
04-19 01:32 19624 cluster cluster DEBUG :Starting prospective KMeans clusterings
04-19 01:32 19624 cluster cluster DEBUG :Trying kmeans(n_clusters=2)
04-19 01:32 19624 cluster cluster DEBUG :Trying kmeans(n_clusters=3)
04-19 01:33 19624 cluster cluster INFO :Best clustering method: kmeans(n_clusters=2) (adjusted
    silhouette == 0.441640409673)
04-19 01:33 19624 cluster <module> INFO :Clustering Complete


Note: shared features does not retain keys from XML and therefore wont always be human readable.
Printing Nessus cluster details

Cluster ID: 0
Silhouette Score: 0.299967117317
IPs: 192.168.0.10, 192.168.0.11

    Shared Features:
    general-purpose
    windows
    445
    cpe:/o:microsoft:windows_7:::professional
    Mon Apr 03 11:25:23 2017
    Microsoft Windows 7 Professional
    9
    5355



Cluster ID: 1
Silhouette Score: 0.0521509960965
IPs: 192.168.0.1, 192.168.0.2

    Shared Features:
    Mon Apr 03 11:25:06 2017
```

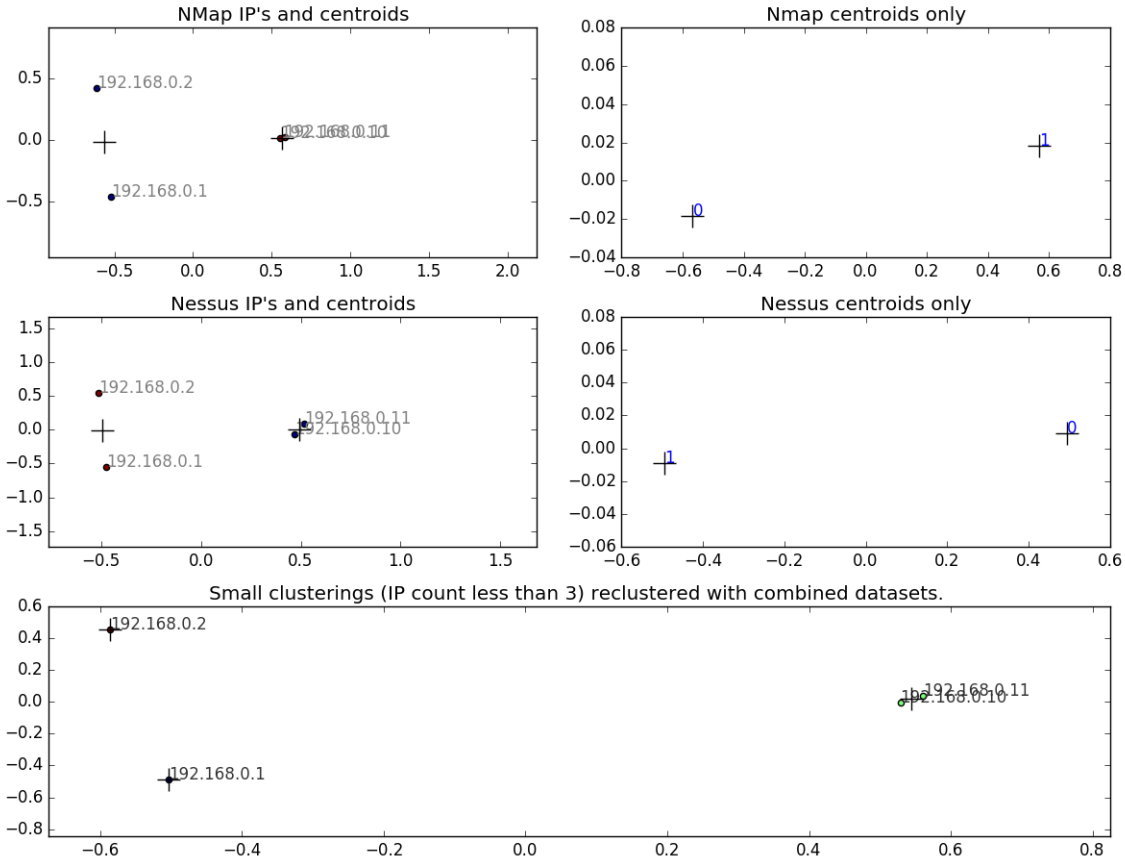
```
    cpe:/o:microsoft:windows
    general-purpose
    windows
    53
    23
    445



cluster ID : amount of IP's
0 : 2
1 : 2



Printing Nmap cluster details

Cluster ID: 0
Silhouette Score: 0.213624573736
IPs: 192.168.0.1, 192.168.0.2

    Shared Features:
    tcp23opentelnet
    tcp42opentcpwrapped
    tcp53opendomain
    tcp80openhttp
    tcp88openkerberos-sec
    tcp135open
    tcp139open
    tcp389open
    tcp445open
    tcp464open
    tcp593open
    tcp636opentcpwrapped
    tcp3268open
    tcp3269opentcpwrapped
    tcp49152open
    tcp49153open
    tcp49154open
    tcp49155open
    tcp49157open
    tcp49158open



Cluster ID: 1
Silhouette Score: 0.66965624561
IPs: 192.168.0.10, 192.168.0.11

    Shared Features:
    tcp135openmsrpcMicrosoft Windows RPC
    tcp139opennetbios-ssnMicrosoft Windows 98 netbios-ssn
    tcp445openmicrosoft-dsMicrosoft Windows 10 microsoft-ds
    tcp49152openmsrpcMicrosoft Windows RPC
    tcp49153openmsrpcMicrosoft Windows RPC
    tcp49154openmsrpcMicrosoft Windows RPC
    tcp49175openmsrpcMicrosoft Windows RPC
    tcp49176openmsrpcMicrosoft Windows RPC



cluster ID : amount of IP's
0 : 2
1 : 2
04-19 01:33 19624 cluster <module> DEBUG :Getting centroids using reduced vectors for Nmap:
04-19 01:33 19624 cluster <module> DEBUG :nclusters: 2
04-19 01:33 19624 clustering get_centroids DEBUG :K = 0
```

```
04-19 01:33 19624 cluster <module> DEBUG :attempting to plot the following centroids:
 [[-0.56736301 -0.01841586]
 [ 0.56736301 0.01841586]]


04-19 01:33 19624 cluster <module> DEBUG :Getting centroids using reduced vectors for Nessus:
04-19 01:33 19624 cluster <module> DEBUG :nclusters: kmeans(n_clusters=2)
04-19 01:33 19624 cluster <module> DEBUG :nclusters: 2
04-19 01:33 19624 clustering get_centroids DEBUG :K = 0

04-19 01:33 19624 cluster <module> DEBUG :attempting to plot the following centroids:
 [[-0.49416507 -0.0089938 ]
 [ 0.49416507 0.0089938 ]]


04-19 01:33 19624 cluster <module> INFO :Nmap Centroids Covariance Matrix:
 [[ 0.64380156 0.02089695]
 [ 0.02089695 0.00067829]]
04-19 01:33 19624 cluster <module> INFO :Nessus Centroids Covariance Matrix:
 [[  4.88398234e-01  8.88884549e-03]
 [  8.88884549e-03  1.61776945e-04]]
04-19 01:33 19624 cluster <module> INFO :distance matrix between centroids using metric for Nmap:
     euclidean :
------- -------
0       1.13532
1.13532 0
------- -------
04-19 01:33 19624 cluster <module> INFO :distance matrix between centroids using metric for Nessus:
     euclidean :
-------- --------
0        0.988494
0.988494 0
-------- --------
04-19 01:33 19624 cluster <module> INFO :IP's from clusters with less than 3 IP's:
 ['192.168.0.1' '192.168.0.10' '192.168.0.11' '192.168.0.2']
04-19 01:33 19624 cluster <module> DEBUG :Loaded 4 vectors with 126 features
04-19 01:33 19624 cluster <module> INFO :Normalizing input and reducing vectors to two dimensions
     with PCA
04-19 01:33 19624 cluster <module> INFO :Resulting single IP vectors:
 [[-0.50300301 -0.49063882]
 [ 0.52888708 -0.00269094]
 [ 0.56066029 0.03834554]
 [-0.58654436 0.45498422]]
04-19 01:33 19624 cluster <module> INFO :distance matrix between centroids of small combined
     clusters: euclidean :
-------- --------- --------- --------
0        1.14144   1.18794   0.949306
1.14144  0         0.0518992 1.20568
1.18794  0.0518992 0         1.22052
0.949306 1.20568   1.22052   0
-------- --------- --------- --------
04-19 01:33 19624 clustering cluster_single_kmeans INFO :Calculating gap statistic value, this can
     take a while...

04-19 01:33 19624 clustering cluster_single_kmeans INFO :gap statistics recommends number of
     clusters: 3

04-19 01:33 19624 clustering cluster_single_kmeans DEBUG :No K value specified, using Gap Statistic

04-19 01:33 19624 cluster <module> INFO :Writing recommended attack IP's to targets.txt for
     exploitation
 {0}
04-19 01:33 19624 display twin DEBUG :twin cluster plot.
04-19 01:33 19624 single_ip_parsing_nmap parse_input DEBUG :found ip 192.168.0.1
04-19 01:33 19624 single_ip_parsing_nmap parse_input DEBUG :found ip 192.168.0.2
04-19 01:33 19624 single_ip_parsing_nmap parse_input DEBUG :found ip 192.168.0.10
04-19 01:33 19624 single_ip_parsing_nmap parse_input DEBUG :found ip 192.168.0.11
```

```
04-19 01:33 19624 display twin DEBUG :{'192.168.0.2': ['"Microsoft Windows 7 SP0 - SP1, Windows
    Server 2008 SP1, Windows 8, or Windows 8.1 Update 1. Prediction accuracy: 100"'], '192.168.0.1':
    [], '192.168.0.10': ['"Microsoft Windows 7 SP0 - SP1, Windows Server 2008 SP1, Windows 8, or
    Windows 8.1 Update 1. Prediction accuracy: 100"'], '192.168.0.11': ['"Microsoft Windows 7 SP0 -
    SP1, Windows Server 2008 SP1, Windows 8, or Windows 8.1 Update 1. Prediction accuracy: 100"']}
```

B.1.2 Figures

The Figure B.1 below shows the example graphing interface output using the configuration mentioned previously in Appendix B.1.



Recommended attack vectors:
192.168.0.2 : "Microsoft Windows 7 SP0 - SP1, Windows Server 2008 SP1, Windows 8, or Windows 8.1 Update 1. Prediction accuracy: 100"
192.168.0.1 : OS detection failed, Didn't receive UDP response
192.168.0.10 : "Microsoft Windows 7 SP0 - SP1, Windows Server 2008 SP1, Windows 8, or Windows 8.1 Update 1. Prediction accuracy: 100"
192.168.0.11 : "Microsoft Windows 7 SP0 - SP1, Windows Server 2008 SP1, Windows 8, or Windows 8.1 Update 1. Prediction accuracy: 100"

Figure B.1: Example of graphing interface GUI from a fictional network to be used in conjunction with text output at B.1.1

B.2 Hacklab Analysis Results

The following results contain outputs of the application created in this thesis when using the University Hacklab network as a dataset for scanning. The scans was obtained after permission was given on the 21st of March 2017. The IP addresses included within the scope is the entire /24 subnet with a mask of 255.255.255.0.

The nmap command used to obtain the scan was as follows:

```
nmap -A -O -oX hacklab.xml 10.0.0.0/24
```

The nessus configuration used was the default policy on an advanced scan. The target IP address configuration of 10.0.0.0/24 was used.

B.2.1 Text Output

The following command was issued to the application in order to achieve the output shown below. It contains the application output in very verbose dual input mode using the hacklab dataset.

```
cluster.py -s automatic -vv -p -t -N -tp "../hacklab
    analyses/hacklab_new.xml" "../hacklab analyses/hacklab_new.nessus"
```

```
04-04 01:22 24060 cluster <module> DEBUG :twin flag enabled
04-04 01:22 24060 cluster <module> DEBUG :tp: ../hacklab analyses/hacklab_new.xml , path:
    ['../hacklab analyses/hacklab_new.nessus']
04-04 01:22 24060 cluster <module> INFO :Vectorizing Stage for Nessus
04-04 01:22 24060 parse_nessus parse_input INFO :Parsing Nessus XML * BETA *
04-04 01:22 24060 parse_nessus parse_input INFO :Done Nessus parsing
no of IP's taken from nessus: 47
04-04 01:22 24060 cluster <module> DEBUG :Loaded 47 vectors with 292 features
04-04 01:22 24060 cluster <module> INFO :Vectorizing complete

04-04 01:22 24060 cluster <module> INFO :Vectorizing Stage for nmap
no of IP's taken from NMAP: 47
04-04 01:22 24060 cluster <module> DEBUG :Loaded 42 vectors with 203 features
04-04 01:22 24060 cluster <module> INFO :Vectorizing complete

04-04 01:22 24060 cluster <module> INFO :Normalising the nessus vectors
04-04 01:22 24060 cluster <module> INFO :Reducing vectors to two dimensions with PCA
04-04 01:22 24060 cluster <module> DEBUG :reduced to 47 vectors with 2 dimensions
04-04 01:22 24060 cluster <module> INFO :Normalising complete

04-04 01:22 24060 cluster <module> INFO :Normalising the nmap vectors
04-04 01:22 24060 cluster <module> INFO :Reducing vectors to two dimensions with PCA
04-04 01:22 24060 cluster <module> DEBUG :reduced to 42 vectors with 2 dimensions
04-04 01:22 24060 cluster <module> INFO :Normalising complete

04-04 01:22 24060 cluster <module> INFO :Clustering Nessus
04-04 01:22 24060 cluster cluster DEBUG :Starting prospective KMeans clusterings
04-04 01:22 24060 cluster cluster DEBUG :Trying kmeans(n_clusters=2)
04-04 01:22 24060 clustering cluster_with_kmeans DEBUG :gap statistics recommends number of clusters:
    39

04-04 01:22 24060 cluster cluster DEBUG :Trying kmeans(n_clusters=3)
```

```
04-04 01:22 24060 cluster cluster DEBUG :Not all clusters are informative (a cluster has 0 shared
    features)
04-04 01:22 24060 cluster cluster DEBUG :Trying kmeans(n_clusters=4)
04-04 01:22 24060 cluster cluster DEBUG :Not all clusters are informative (a cluster has 0 shared
    features)
04-04 01:22 24060 cluster cluster DEBUG :Trying kmeans(n_clusters=5)
04-04 01:22 24060 cluster cluster DEBUG :Not all clusters are informative (a cluster has 0 shared
    features)
04-04 01:22 24060 cluster cluster DEBUG :Trying kmeans(n_clusters=6)
04-04 01:22 24060 cluster cluster DEBUG :Not all clusters are informative (a cluster has 0 shared
    features)
04-04 01:22 24060 cluster cluster DEBUG :Trying kmeans(n_clusters=7)
04-04 01:22 24060 cluster cluster DEBUG :Trying kmeans(n_clusters=8)
04-04 01:22 24060 cluster cluster INFO :Best clustering method: kmeans(n_clusters=7) (adjusted
    silhouette == 0.408249368504)
04-04 01:22 24060 cluster <module> INFO :Clustering Complete


04-04 01:22 24060 cluster <module> INFO :Clustering Nmap
04-04 01:22 24060 cluster cluster DEBUG :Starting prospective KMeans clusterings
04-04 01:22 24060 cluster cluster DEBUG :Trying kmeans(n_clusters=2)
04-04 01:23 24060 clustering cluster_with_kmeans DEBUG :gap statistics recommends number of clusters:
    24

04-04 01:23 24060 cluster cluster DEBUG :Trying kmeans(n_clusters=3)
04-04 01:23 24060 cluster cluster DEBUG :Not all clusters are informative (a cluster has 0 shared
    features)
04-04 01:23 24060 cluster cluster DEBUG :Trying kmeans(n_clusters=4)
04-04 01:23 24060 cluster cluster DEBUG :Not all clusters are informative (a cluster has 0 shared
    features)
04-04 01:23 24060 cluster cluster DEBUG :Trying kmeans(n_clusters=5)
04-04 01:23 24060 cluster cluster DEBUG :Trying kmeans(n_clusters=6)
04-04 01:23 24060 cluster cluster INFO :Best clustering method: kmeans(n_clusters=5) (adjusted
    silhouette == 0.411583679893)
04-04 01:23 24060 cluster <module> INFO :Clustering Complete


Note: shared features does not retain keys from XML and therefore wont always be human readable.
Printing Nessus cluster details

Cluster ID: 0
Silhouette Score: 0.220046730181
IPs: 10.0.0.24, 10.0.0.26, 10.0.0.27, 10.0.0.30, 10.0.0.33, 10.0.0.36, 10.0.0.39, 10.0.0.40,
    10.0.0.41, 10.0.0.42, 10.0.0.43, 10.0.0.46, 10.0.0.47, 10.0.0.48, 10.0.0.50, 10.0.0.51,
    10.0.0.53, 10.0.0.54, 10.0.0.55, 10.0.0.56, 10.0.0.58, 10.0.0.64, 10.0.0.65, 10.0.0.66,
    10.0.0.70, 10.0.0.76, 10.0.0.79, 10.0.0.83, 10.0.0.84

    Shared Features:
    cpe:/o:microsoft:windows_7:::professional
    cpe:/a:openbsd:openssh:7.1 -> OpenBSD OpenSSH 7.1
    SSH-2.0-OpenSSH_7.1p1 Microsoft_Win32_port_with_VS
    general-purpose
    Microsoft Windows 7 Professional
    windows
    445



Cluster ID: 1
Silhouette Score: 0.18063333287
IPs: 10.0.0.32, 10.0.0.37, 10.0.0.77, 10.0.0.81

    Shared Features:
    1



Cluster ID: 2
```

```
Silhouette Score: 0.234851036884
IPs: 10.0.0.28, 10.0.0.34, 10.0.0.35, 10.0.0.69, 10.0.0.74, 10.0.0.78, 10.0.0.80, 10.0.0.85

    Shared Features:
    general-purpose
    cpe:/o:linux:linux_kernel:3.10
    cpe:/o:linux:linux_kernel:3.13
    cpe:/o:linux:linux_kernel:4.2
    cpe:/o:linux:linux_kernel:4.8
    cpe:/a:openbsd:openssh:7.2
    SSH-2.0-OpenSSH_7.2
    Linux Kernel 3.10\\nLinux Kernel 3.13\\nLinux Kernel 4.2\\nLinux Kernel 4.8
    1
    linux



Cluster ID: 3
Silhouette Score: 0.228689064888
IPs: 10.0.0.185, 10.0.0.82

    Shared Features:
    general-purpose
    cpe:/a:openbsd:openssh:7.2
    cpe:/o:linux:linux_kernel
    1
    linux
    cpe:/o:canonical:ubuntu_linux:16.04
    SSH-2.0-OpenSSH_7.2p2 Ubuntu-4ubuntu2.1
    Linux Kernel 4.4 on Ubuntu 16.04 (xenial)



Cluster ID: 4
Silhouette Score: -0.00647458529864
IPs: 10.0.0.1, 10.0.0.7

    Shared Features:
    443
    3



Cluster ID: 5
Silhouette Score: 1.0
IPs: 10.0.0.5

    Shared Features:
    cpe:/o:microsoft:windows_7:::professional
    general-purpose
    Microsoft Windows 7 Professional
    windows
    445
    1490105876
    Hacklab2
    Tue Mar 21 14:17:56 2017
    Tue Mar 21 14:15:50 2017
    HACKLAB2
    10.0.0.5
    9
    00:25:b3:28:d9:14
    49158
    5355



Cluster ID: 6
Silhouette Score: 1.0
```

```
IPs: 10.0.0.3

    Shared Features:
    general-purpose
    7
    443
    445
    cpe:/a:openbsd:openssh:7.2
    cpe:/o:linux:linux_kernel
    linux
    Tue Mar 21 14:15:50 2017
    67
    cpe:/o:canonical:ubuntu_linux:16.04
    cpe:/a:samba:samba:4.3.11
    cpe:/a:apache:http_server:2.4.18
    10.0.0.3
    HACKLAB1
    Apache Multiviews Arbitrary Directory Listing: Upgrade to Apache version 1.3.22 or later.
         Alternatively, as a workaround, disable Multiviews.
    1490105952
    SSH-2.0-OpenSSH_7.2p2 Ubuntu-4ubuntu2.1
    Linux Kernel 4.4 on Ubuntu 16.04 (xenial)
    hacklab1
    Tue Mar 21 14:19:12 2017
    00:25:b3:1f:07:ae
    80



cluster ID : amount of IP's
0 : 29
1 : 4
2 : 8
3 : 2
4 : 2
5 : 1
6 : 1



Printing Nmap cluster details

Cluster ID: 0
Silhouette Score: 0.528486221388
IPs: 10.0.0.24, 10.0.0.26, 10.0.0.27, 10.0.0.30, 10.0.0.33, 10.0.0.36, 10.0.0.39, 10.0.0.41,
     10.0.0.42, 10.0.0.43, 10.0.0.46, 10.0.0.47, 10.0.0.48, 10.0.0.50, 10.0.0.51, 10.0.0.53,
     10.0.0.54, 10.0.0.55, 10.0.0.56, 10.0.0.58, 10.0.0.64, 10.0.0.65, 10.0.0.66, 10.0.0.70,
     10.0.0.76, 10.0.0.79, 10.0.0.83, 10.0.0.84

    Shared Features:
    tcp22opensshprotocol 2.0
    tcp135openmsrpcMicrosoft Windows RPC
    tcp139opennetbios-ssn
    tcp443openhttpVMware VirtualCenter Web service
    tcp443openhttpssl-certlocalityNamePalo Alto
    tcp443openhttpssl-certorganizationalUnitNameVMware
    tcp443openhttpssl-certcommonNameVMware
    tcp443openhttpssl-certemailAddressnone@vmware.com
    tcp443openhttpssl-certcountryNameUS
    tcp443openhttpssl-certtypersa
    tcp443openhttpssl-certbits2048
    tcp443openhttpssl-certnotAfter2017-08-03T11:59:23+00:00
    tcp443openhttpssl-certnotBefore2016-08-03T11:59:23+00:00
    tcp443openhttpssl-certmd5ad6e5f01314d1518b4051099cd81a1f0
    tcp443openhttpssl-certsha183a0d223f33891ba5192d94c26c44e05cfe72a51
    tcp443openhttpssl-certpem-----BEGIN
         CERTIFICATE-----\\nMIIEEjCCAvqgAwIBAgIJAP9VV+H4OSLcMA0GCSqGSIb3DQEBCwUAMGMxCzAJBgNV\\nBAYTAlVTMRIwEAYDVQQHDAlQ
         CERTIFICATE-----\\n
```

```
    tcp445opennetbios-ssn


Cluster ID: 1
Silhouette Score: 1.0
IPs: 10.0.0.28, 10.0.0.34, 10.0.0.35, 10.0.0.40, 10.0.0.69, 10.0.0.74, 10.0.0.78, 10.0.0.80, 10.0.0.85

    Shared Features:
    tcp22opensshOpenSSH7.2protocol 2.0


Cluster ID: 2
Silhouette Score: -0.222222631321
IPs: 10.0.0.1, 10.0.0.7

    Shared Features:
    tcp443open
    tcp80openhttp


Cluster ID: 3
Silhouette Score: -0.248345190601
IPs: 10.0.0.3, 10.0.0.5

    Shared Features:
    tcp139opennetbios-ssn
    tcp445opennetbios-ssn


Cluster ID: 4
Silhouette Score: 1.0
IPs: 10.0.0.185

    Shared Features:
    tcp22opensshprotocol
        2.0SF-Port22-TCP:V=6.47\%I=7\%D=3/21\%Time=58D136BB\%P=i686-pc-windows-windows\%r(NULL,29,"SSH-2\\\\.0-OpenSSH


cluster ID : amount of IP's
0 : 28
1 : 9
2 : 2
3 : 2
4 : 1
04-04 01:23 24060 cluster <module> DEBUG :Getting centroids using reduced vectors for Nmap:
04-04 01:23 24060 cluster <module> DEBUG :nclusters: 5
04-04 01:23 24060 cluster <module> DEBUG :attempting to plot the following centroids:
 [[-0.34575755 0.05203286]
 [ 0.93722406 0.14279432]
 [ 0.06086076 -0.45066068]
 [ 0.31235087 -0.80310291]
 [ 0.49977173 -0.23454181]]


04-04 01:23 24060 cluster <module> DEBUG :Getting centroids using reduced vectors for Nessus:
04-04 01:23 24060 cluster <module> DEBUG :nclusters: kmeans(n_clusters=5)
04-04 01:23 24060 cluster <module> DEBUG :nclusters: 7
04-04 01:23 24060 cluster <module> DEBUG :attempting to plot the following centroids:
 [[-0.32465071 0.04241239]
 [ 0.73854713 0.28508362]
 [ 0.42526042 -0.51413803]
 [ 0.28150282 -0.11457564]
 [ 0.55611277 -0.12418121]
```

```
 [-0.06618649 -0.0772538 ]
 [ 0.23895499 -0.50694214]]


04-04 01:23 24060 cluster <module> INFO :Nmap Centroids Covariance Matrix:
 [[ 0.23001336 0.02479788]
 [ 0.02479788 0.14788991]]
04-04 01:23 24060 cluster <module> INFO :Nessus Centroids Covariance Matrix:
 [[ 0.13216597 0.00459265]
 [ 0.00459265 0.08221746]]
04-04 01:23 24060 cluster <module> INFO :distance matrix between centroids using metric for Nmap:
     euclidean :
-------- -------- -------- -------- --------
0        1.28619  0.64656  1.07906  0.892774
1.28619  0        1.0584   1.13366  0.577708
0.64656  1.0584   0        0.43297  0.489234
1.07906  1.13366  0.43297  0        0.598655
0.892774 0.577708 0.489234 0.598655 0
-------- -------- -------- -------- --------
04-04 01:23 24060 cluster <module> INFO :distance matrix between centroids using metric for Nessus:
     euclidean :
-------- -------- -------- -------- -------- -------- --------
0        1.09054  0.933871 0.626153 0.89638  0.284822 0.787046
1.09054  0        0.858431 0.607138 0.448085 0.882544 0.936428
0.933871 0.858431 0        0.424637 0.411325 0.657562 0.186444
0.626153 0.607138 0.424637 0        0.274778 0.349687 0.394667
0.89638  0.448085 0.411325 0.274778 0        0.624066 0.497087
0.284822 0.882544 0.657562 0.349687 0.624066 0        0.527014
0.787046 0.936428 0.186444 0.394667 0.497087 0.527014 0
-------- -------- -------- -------- -------- -------- --------
04-04 01:23 24060 cluster <module> INFO :IP's from clusters with less than 3 IP's:
 ['10.0.0.1' '10.0.0.185' '10.0.0.3' '10.0.0.5' '10.0.0.7' '10.0.0.82']
04-04 01:23 24060 cluster <module> DEBUG :Loaded 6 vectors with 495 features
04-04 01:23 24060 cluster <module> INFO :Normalizing input and reducing vectors to two dimensions
     with PCA
04-04 01:23 24060 cluster <module> INFO :Resulting single IP vectors:
 [[-0.57810661 0.35576052]
 [ 0.23495851 -0.64514061]
 [ 0.0701816 -0.41400474]
 [ 0.49713564 0.35728938]
 [-0.57362822 -0.09504716]
 [ 0.34945907 0.4411426 ]]
04-04 01:23 24060 cluster <module> INFO :distance matrix between centroids of small combined
     clusters: euclidean :
-------- -------- -------- -------- -------- --------
0        1.28953  1.00639  1.07524  0.45083  0.931487
1.28953  0        0.283858 1.03615  0.977965 1.0923
1.00639  0.283858 0        0.881581 0.718488 0.899596
1.07524  1.03615  0.881581 0        1.16239  0.169823
0.45083  0.977965 0.718488 1.16239  0        1.06752
0.931487 1.0923   0.899596 0.169823 1.06752  0
-------- -------- -------- -------- -------- --------
04-04 01:23 24060 clustering cluster_single_kmeans DEBUG :gap statistics recommends number of
     clusters: 3

04-04 01:23 24060 display twin DEBUG :twin cluster plot.
```

B.2.2 Figures

Figure B.2 shows the application output in standard mode using the Nessus dataset from the hacklab. The command used is as follows:

```
cluster.py -s automatic -vv -p -N "../hacklab analyses/hacklab_new.nessus"
```

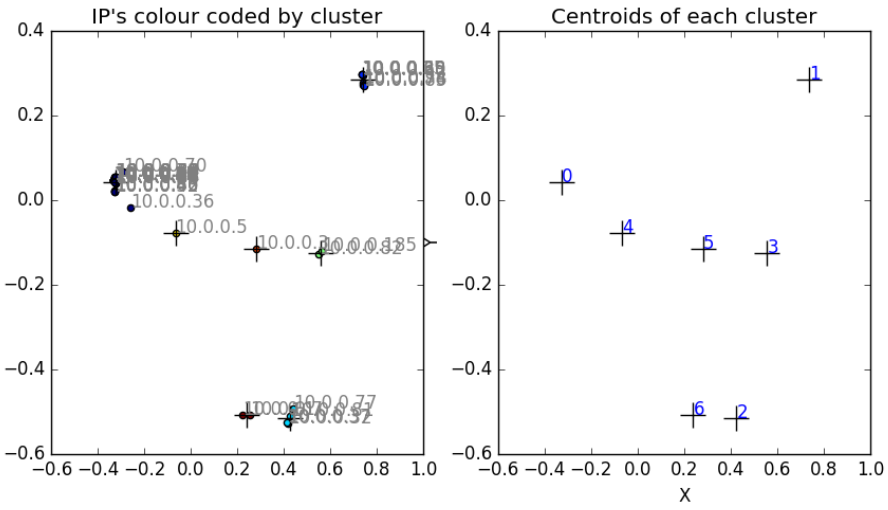


Figure B.2: Example graphing interface output using the the Abertay University Hacklab Nessus scan as a dataset.

The Figure B.3 below shows output to that similar of the Figure B.2 above however this used the Nmap dataset as appose to the Nessus one used in that figure. The command used to acheive this result is as follows:

```
cluster.py -s automatic -vv -p "../hacklab analyses/hacklab_new.xml"
```

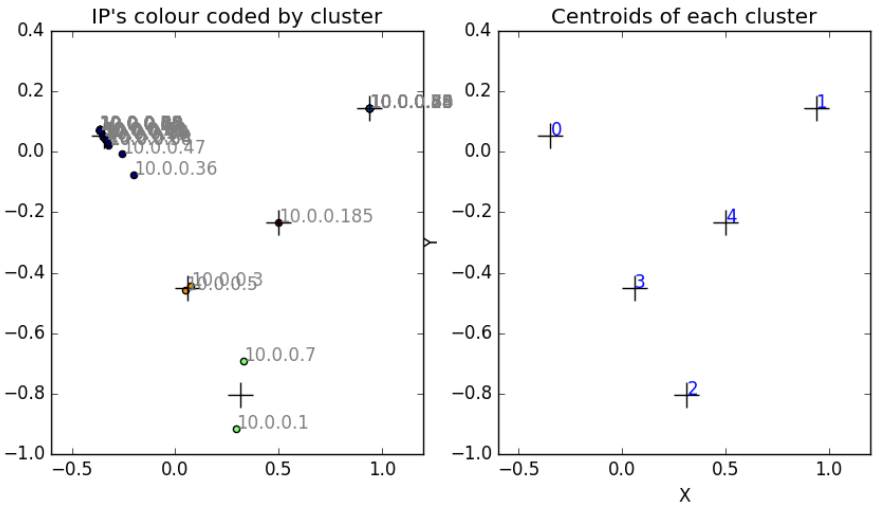
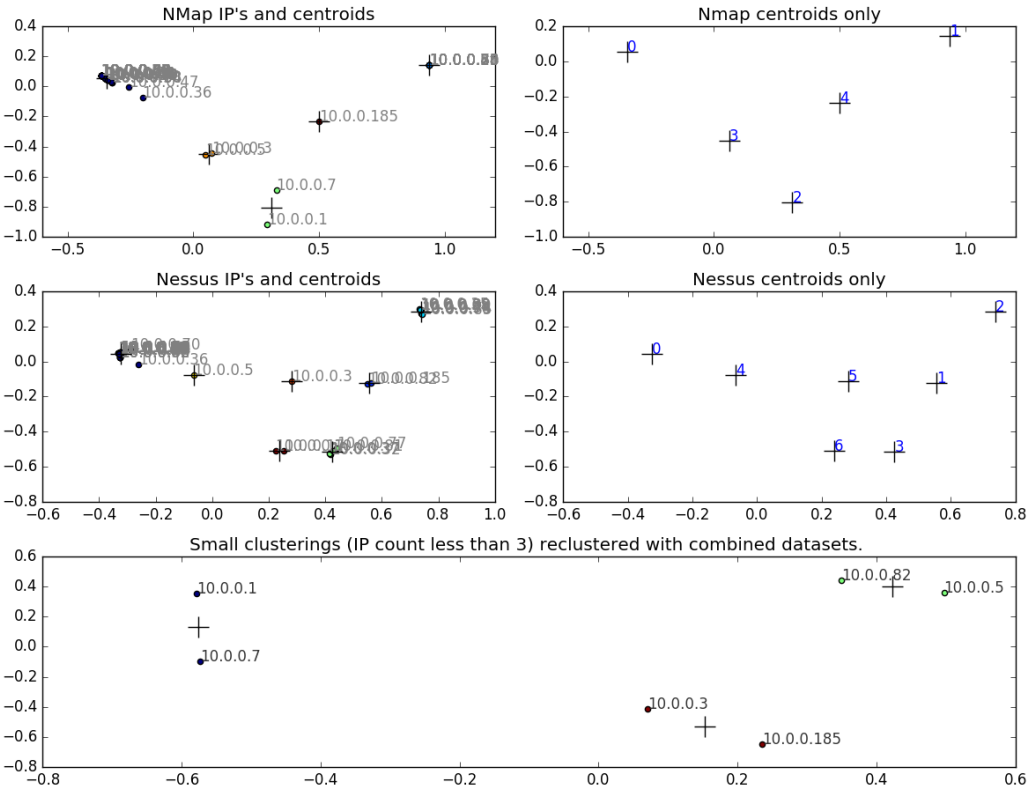


Figure B.3: Example graphing interface output using the the Abertay University Hacklab Nmap scan as a dataset.

Figure B.4 below shows the graphing interface opened after the completion of the clustering algorithm. This was acheived using dual mode and the command use can be found above in B.2.1.



Recommended attack vectors:
10.0.0.5 : "Microsoft Windows 7 SP0 - SP1, Windows Server 2008 SP1, or Windows 8. Prediction accuracy: 100"
10.0.0.7 : "HP iLO 2 remote management interface. Prediction accuracy: 100"
10.0.0.185 : "Linux 3.11 - 3.14. Prediction accuracy: 100"
10.0.0.1 : "Linux 2.6.18 - 2.6.22. Prediction accuracy: 86"
10.0.0.3 : "Linux 3.11 - 3.14. Prediction accuracy: 100"

Figure B.4: Example graphing interface output using both scans of Abertay University Hacklab and running the application in dual mode.

Bibliography

Black, J. & Zernik, U. (1994), 'Method for natural language data processing using morphological and part-of-speech information'. US Patent 5,331,556.
URL: *http://www.google.com/patents/US5331556*

Blackhat USA, C. (2016), 'Nmap cluster'.
URL: *https://github.com/CylanceSPEAR/NMAP-Cluster*

DeepMind (2016), 'Alphago'.
URL: *https://deepmind.com/alpha-go*

Karpathy, A. (2011), 'Lessons learned from manually classifying cifar-10'.
URL: *http://karpathy.github.io/2011/04/27/manually-classifying-cifar10/*

Kohavi, R. & Provost, F. (1998), 'Glossary of terms', *Machine Learning* **30**(2-3), 271–274.

Kurenkov, A. (2016), 'A 'brief' history of game ai up to alphago, part 1'.
URL: *http://www.andreykurenkov.com/writing/a-brief-history-of-game-ai/*

LeCun, Y., Bengio, Y. & Hinton, G. (2015), 'Deep learning', *Nature* **521**(7553), 436–444.

Matthias Feurer, Aaron Klein, F. H. (2016), 'Contest winner: Winning the automl challenge with auto-sklearn'.
URL: *http://www.kdnuggets.com/2016/08/winning-automl-challenge-auto-sklearn.html*

Mattsson, N. (2016), 'Classification performance of convolutional neural networks'.

Mayo, M. (2017), 'The current state of automated machine learning'.
URL: *http://www.kdnuggets.com/2017/01/current-state-automated-machine-learning.html*

Munoz, A. (2012), 'Machine learning and optimization@ONLINE'.
URL: *https://www.cims.nyu.edu/ munoz/files/ml_optimization.pdf*

Patro, S. & Sahu, K. K. (2015), 'Normalization: A preprocessing stage', *arXiv preprint arXiv:1503.06462* .

Rossant, C. (2014), *Introduction to Machine Learning in Python with scikit-learn*, Packt Publishing.
URL: *http://ipython-books.github.io/featured-04/*

Rousseeuw, P. J. (1987), 'Silhouettes: a graphical aid to the interpretation and validation of cluster analysis', *Journal of computational and applied mathematics* **20**, 53–65.

Simon, P. (2013), *Too Big to Ignore: The Business Case for Big Data*, Wiley and SAS Business Series, Wiley.
URL: *https://books.google.co.uk/books?id=1ekYIAoEBrEC*

Tibshirani, R., Walther, G. & Hastie, T. (2001), 'Estimating the number of clusters in a data set via the gap statistic', *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* **63**(2), 411–423.

Tipping, M. E. & Bishop, C. M. (1999), 'Mixtures of probabilistic principal component analyzers', *Neural computation* **11**(2), 443–482.

Valiant, L. G. (1984), 'A theory of the learnable', *Communications of the ACM* **27**(11), 1134–1142.