USING VARIATIONAL AUTOENCODERS WITH MACHINE LEARNING ALGORITHMS IN CYBER SECURITY APPLICATIONS

A DISSERTATION SUBMITTED TO MANCHESTER METROPOLITAN UNIVERSITY
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2022

By
Thomas Taylor
Department of Computing and Mathematics

Contents

List of Tables

List of Figures

Abstract

Often complex datasets will have a large number of features for each of its samples. Sometimes, this can have a negative effect on the performance of models trained on the raw data. By reducing the number of features this problem can be avoided. However, this may cause a loss of information. One method to mitigate this is by using a type of unsupervised neural network structure called autoencoders. Autoencoders can be used to generate a reduced feature space which the models can then be trained on. This paper uses Convolutional Variational Autoencoders in order to create these latent features and then determines their effectiveness of improving performance of machine learning classifier models.

This reports looks at two cybersecurity datasets: The Microsoft Malware Challenge 2015 Dataset, and the NSL-KDD dataset with both a multiclass and binary configuration. VAEs were trained on each dataset and the latent features generated from the encoder were extracted and used to train various ML classifiers. The ML classifiers that were used are Gaussian Naïve Bayesian Networks, Support Vector Machines with a RBF kernel, Decision Trees, and Dense Neural Networks. Once the networks were trained, performance metrics were generated. Namely, accuracy, precision, recall, F1 measure, and Mathews Correlation Coefficient. In order to properly gauge the effectiveness of the VAEs, PCA thresholding was used as an alternative dimensional reduction technique, and each of the models were trained on those features. The resulting performing metrics were compared against each other.

The maximum performing models for the 3 datasets were: For Malware Multiclass a SVM RBF model with no DR techniques. This produced 97.8381% accuracy, an F1 of 0.978586, and a MCC of 0.973872 For NSL-KDD Binary a SVM RBF with a PCA threshold of 0.92, and a VAE with 6 latent features. This produced 82.8070% accuracy, an F1 of 0.830461, and a MCC of 0.682102 For NSL-KDD Multiclass a Decision Tree using gini entropy, using a `best` splitter, with a MSS of 8 and a MSL of 4. This produced 76.2598% accuracy, 0.723142 F1, and 0.653463 MCC.

Declaration

No part of this project has been submitted in support of an application for any other degree or qualification at this or any other institute of learning. Apart from those parts of the project containing citations to the work of others, this project is my own unaided work. This work has been carried out in accordance with the Manchester Metropolitan University research ethics procedures, and has received ethical approval number 34865.

Signed:

TTaylor

Date: June 5, 2022

Acknowledgements

I would like to thank my project supervisor Dr Amna Eleyan for providing invaluable support and guidance. I would also like to thank my parents.

Abbreviations

ASM	Assembly
CSV	Comma Seperated Values
LFD	Latent Feature Dimention
ML	Machine Learning
MLE	Maximum Likelihood Estimation
MSL	Minimum Samples Leaf
MSS	Minimum Samples Split
NN	Neural Network
PCA	Principal Component Analysis
RBF	Radial Basis Function
SDG	Stochastic gradient descent
SVM	Support Vector Machine
VAE	Variational Autoencoder

Chapter 1

Introduction

1.1 Project Overview

Cybersecurity is an extremely important field in modern science. As the number of people who use computers continue to increase so will the amount of software being developed. Both malicious and benign. In order to protect new and existing systems precautions need to be developed. Developing general purpose solutions to identify malicious attackers is one method to solve this issue. This report looks at network intrusion and malware datasets and constructs models which are able to classify different types of attacks.

Currently there is a large interest in the field of machine learning. It is used in all sorts of applications from game theory (DeepMind 2016) to healthcare (Ahmad, Eckert, and Teredesai 2018). Specifically there has been large amounts of research into the field of neural networks (NNs). NNs provide a novel way of solving non-trivial classification and regression problems. By varying the format of the data and the shape of the network, NNs are able to solve a massive range of problems. This report will look into how a specific form of NNs, variational autoencoders, are able to increase the performance of other machine learning algorithms.

1.2 Aims and Objectives

Aims

This study attempts to determines the effectiveness of using variational autoencoders to produce latent feature representations of data in order to increase the accuracy of machine learning algorithms. Specifically, neural networks, Bayesian networks, support vector machines, and decision trees are trained and tested with malware and intrusion detection datasets.

Objectives

- Terms of Reference - Providing a more detailed scope and description of the project
- Data Collation - Collation and preparation of the malware and intrusion detection datasets including basic analysis and preprocessing
- Data Analysis & Software Construction - Training and testing of the different machine learning methods including parameter tuning
- Final Report - Report encompassing all work done and information gathered

Motivation/Impact

There is a large drive in cybersecurity to develop new methods to detect malicious parties. This paper looks into two fields of cybersecurity and investigates novel methods to attempt to increase performance of existing machine learning classifiers. If this paper determines that VAEs have a positive influence on the performance of machine learning classifiers, this may provide increased performance to malware and intrusion detection software.

Contribution

This project explores a relatively unused aspect of convolutional variational autoencoders. Existing machine learning classifier algorithms are trained with latent features generated by VAEs to discover if this application of VAEs improves performance. This technique has never been used before with the two datasets this paper uses. The performance of this VAE technique is also compared against the effect of using no dimensional reduction techniques instead, using PCA thresholding, and also using a combination of VAE and PCA.

1.3 Tools and Timeline

`Python` (Foundation 2016) is the main tool that is used to run this experiment. `pandas` (NumFOCUS 2021) and `numpy` (NumPy 2021) are used to store the datasets in memory in order to perform analysis on them. `TensorFlow` (Martıén Abadi et al. 2015) is used to construct any neural networks including the VAEs and dense neural networks. `scikit learn` (Pedregosa et al. 2011) is used to create the ML models that are not neural networks. i.e. Bayesian networks, support vector machines, and decision trees. `Matplotlib` (Hunter 2007), `seaborn` (Waskom 2021), and `Jupyter` (Kluyver et al. 2016) are used to analyse the generated results and create visualisations. `Git` (Chacon and Straub 2014) is used for version control.

1.4 Report Structure

This report is structured like so:

Chapter 1 - Introduction An overview of the scope, structure, and aims of the project as well as a list of the technologies used.

Chapter 2 - Literature Review An explanation of the technologies that go into this project including descriptions of the algorithms behind them. This section also includes an investigation into the works done previously in this field and a list of improvements that this experiment takes from them.

Chapter 3 - Datasets A description of the datasets used in this experiment and a description of the preprocessing performed on them.

Chapter 4 - Experimental Methodology A description of the methodology and the structure of the experiment. This includes a description of the application that ran this experiment.

Chapter 5 - Experimental Implementation Implementation details on technologies used and parameters chosen.

Chapter 6 - Experimental Results Analysis and insight into the results of the experiment including visualisations.

Chapter 7 - Conclusion A summary of the papers results as well as an analysis on the completion of the objectives and aims of this report. Also includes a discussion of relevant work that could be carried on as an extension of this paper.

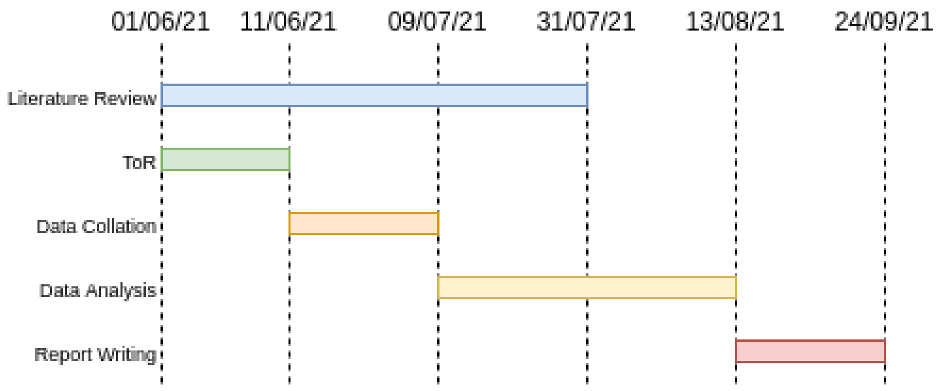

Figure 1.1: Gantt chart detailing the timeline of the project

Chapter 2

Literature Review

In order for the research this report conducts to be understood, it is imperative that the techniques and algorithms that are used are fully described. This section covers all the non-trivial content that is used.

2.1 Intrusion Attacks

Intrusion attacks refer to an malicious attempt to affect an internal system from an outside network. This can come in many forms, from denial of service (DoS), to smurfing, mailbombs, and cross site scripting. As well as these well known attack, new vulnerabilities are always being discovered. As a result, there is a large drive to develop general purpose algorithms that are able to identify attacks. Section 2.6 describes previous papers that look at various methods that try to classify traffic into malicious and benign packets, and also into the specific type of attack that is happening. Section 3.1 describes NSL-KDD, one of the most common intrusion detection datasets. Of the classes listed in tables 3.1 and 3.2, there are four main meta classes (Saporito 2019).

DoS Denial of service is an attack that attempts to stop incoming traffic to the network by sending a large enough number of request so that the system is unable to cope. This causes the system to either take too long for any single request due to the amount to processing that is taking place, or it shuts down from overloading.

Probe Probe attacks attempt to gain knowledge about a system. This can possibly be done by sending odd queries, or by sending queries that attempt to observe network traffic.

U2R User to root tries to gain access to a system by first logging in as a regular user, and then using privilege escalation to become a super/root user.

R2L Remote to local attacks attempt to gain access to a system from a remote machine.

2.2 Malware

Malware is a type of malicious software that attempts to control and spread to other computers. In the Microsoft malware challenge dataset (see section 3.2) there are 9 distinct types that are classified.

Kelihos_ver1 & Kelihos_ver3 are types of botnet (Ortloff 2012). They try to create a decentralised network of infected computers which can then be used to steal and mine bitcoin, and collect email addresses by searching the hard drive and intercepting email traffic.

Lollipop displays pop-up advertisements as you browse the internet based of keywords that are entered into certain search engines (Microsoft 2013a). It infects PCs by entering itself into third-party software bundles and advertising itself as a discount and offer viewer.

Ramnit is a family of multi-component malware that spreads to removable drives, steals sensitive information such as saved banking and FTP credentials and browser cookies. The malware may also open a backdoor to await instructions from a remote attacker. (Microsoft 2011a)

Obfuscator.ACY is a set of different malware that are hidden behind an obfuscation layer. This attempts to hide the active malware from the security software. (Microsoft 2014)

Gatak is a type of trojan. It can pose as an update to legitimate applications or arrive as part of a key generator application. After installing itself on the host machine it then collects information about the PC and sends it back to a remote server. (Microsoft 2013b)

Tracur is a family of trojans that redirect web search queries to malicious URLs, download and run and other files, including other malware, and let backdoor access and remote control. (Microsoft 2011b)

Vundo is a family of programs that deliver “out of context” pop-up advertisements. They can also download and run files. They often spread as DLL files and install on PCs as a Browser Helper Object (BHO) without consent. This family also uses advanced techniques to avoid detection and removal. (Microsoft 2009)

Simda is a multi-component trojan that downloads and executes arbitrary files. These files may include additional malware. (Microsoft 2010)

2.3 Machine Learning

Mitchell et al. 1997

> The field of machine learning is concerned with the question of how to construct computer programs that automatically improve with experience.

Machine learning (or ML) comes in many forms but generally algorithms try to “learn” from the data provided in order to discover trends and patterns in data that may not be visible to a human looking through the data manually using conventional statistical methods. Typically these algorithms contain smaller decisions and calculations, that when combined and layered can produce complex and accurate classifications methods.

2.3.1 Bayes’ Theorem and Bayesian networks

Bayesian networks attempt to classify data using a probabilistic model. They use Bayes’ theorem in order to determine what the most likely class is of a data point given the data’s values (Mitchell et al. 1997).

Bayes’ theorem is defined as

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \tag{2.1}$$

Where A and B are events and $P(B)! = 0$ and P denotes the probability of an event (Stuart 1994).

This equation describes the relationship between 2 non-independent events. In the case of machine learning the following example could happen: Say that even A is the fact that a hospital patient has the flu. Without any information this chance is very small, 0.001% (The percentage of patients who had the flu last year). Say however event B is observed which is that the patient has a high fever. With this extra information Bayes' theorem says that the probability that a patient has the flu given that they have a high fever is much higher (say 50%).

This idea can be expanded to observe multiple events to increase the accuracy of the prediction. Stringing these observations together gives a Bayesian network, where the probability of any one event can be found by observing the events that effect it.

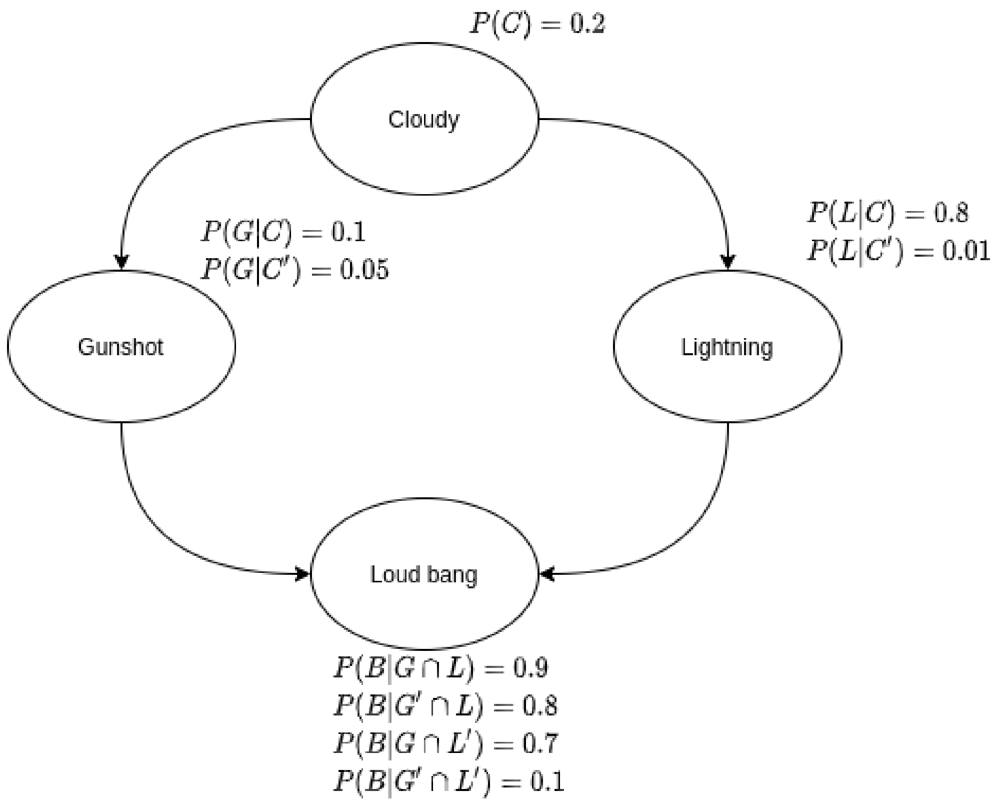

Figure 2.1: Example of a typical Bayesian network

In figure 2.1 an example Bayesian network is shown. From this network it can be show, by observing events like $P(C)$, $P(G)$, and $P(L)$ the probability of $P(B)$ is able to be calculated with more accuracy.

2.3.2 Decision Trees

From scikit-learn 2021a:

"Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation."

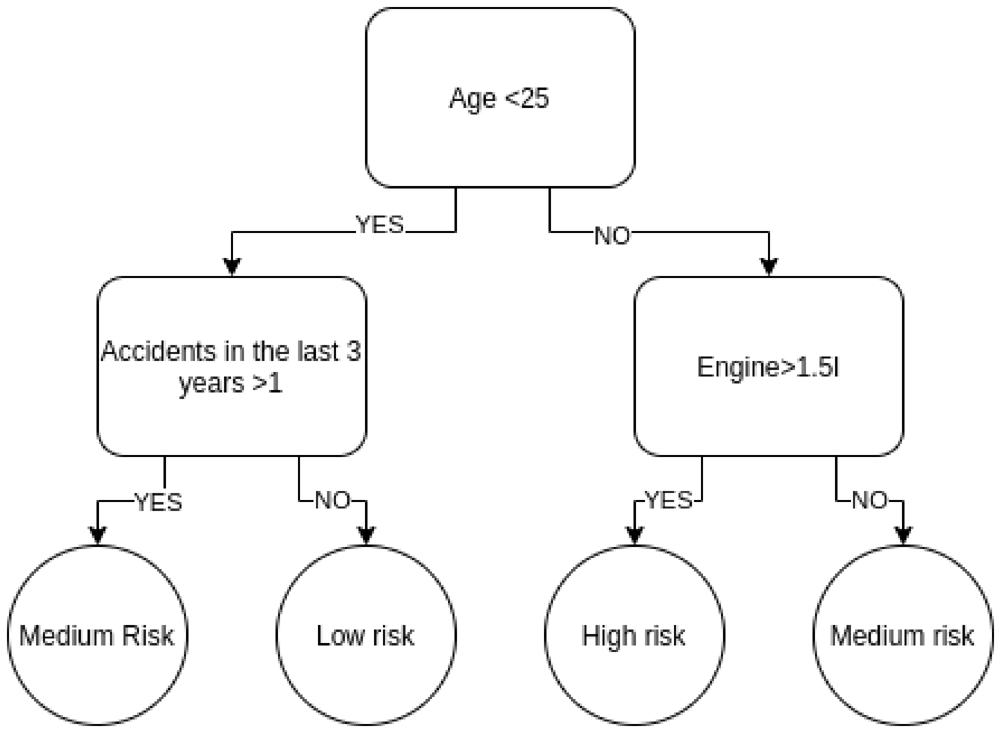

Figure 2.2: Example Decision Tree to classify car insurance risk factor

In other words, Decision Trees can be seen as constructed models that simulate a flowchart like structure which aims to classify samples through a system of micro decisions. In order to construct these models, at each node a decision needs to be made on what parameter to split on. In this paper an optimized form of the CART algorithm (Classification And Regression Trees) is used. CART uses "information gain" in order to gauge the most effective component to split on at each node. Information gain is defined as (Berrar and Dubitzky 2013):

$$IG(T, a) = H(T) - H(T|a) \tag{2.2}$$

Where T refers to the state of the model, a refers to a variable set to a constant, and H calculates the information entropy. This in one way measures the purity of the new sets of classes that are created from splitting on the variable a.

2.3.3 Neural Networks

Neural networks are a type of machine learning inspired by a brain's neurons. A networks is made up of individual neurons. Each neuron can take some input, put that input through some sort of weighted function, and then output a value. These networks get far too big for a human to manually set the parameters, so they are trained via a technique called backpropagation.

Each neuron is defined as a function that takes in any number of inputs and produces a single output. Each of the inputs are give a weight, and the output is calculated by combining the sum of the inputs multiplied by their respective weights and then passed into a sigmoid function with a bias. This function is visualised in figure 2.3.

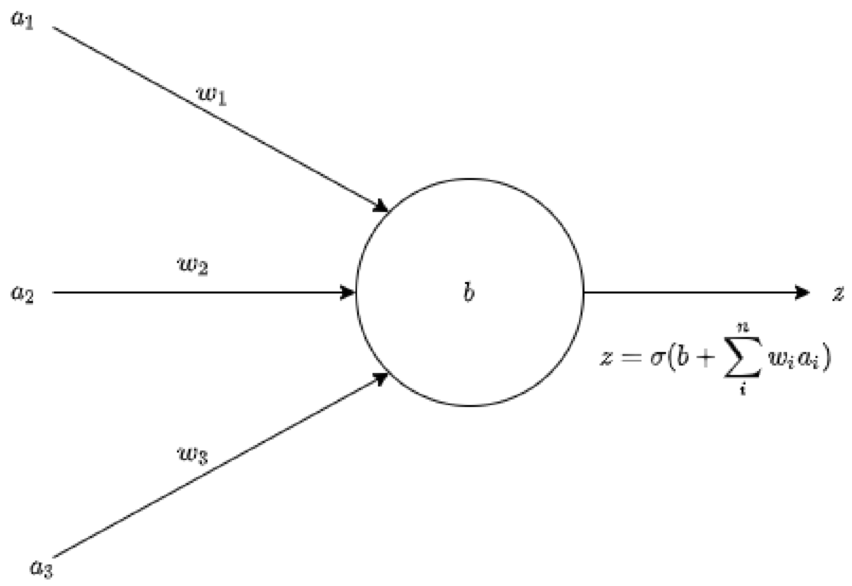

Figure 2.3: Representation of a neuron

Typically neural networks are structured in layers, with each layer being fully connected to the next. Below is an example of a 3 layer networks each with 3 neurons per network. Each of the neurons have three inputs weights and each neuron has a static bias, meaning that in total there are 24 constants to tune for this relatively simple network.

In the case of the above network, the input item has 3 features (e.g. weight, height, age) and there are 3 possible classes to choose from for classification (e.g. athlete, office worker, retail). Often for classification problems, the final layer of a neural network will have a sigmoid function attached to the output. This will limit the output

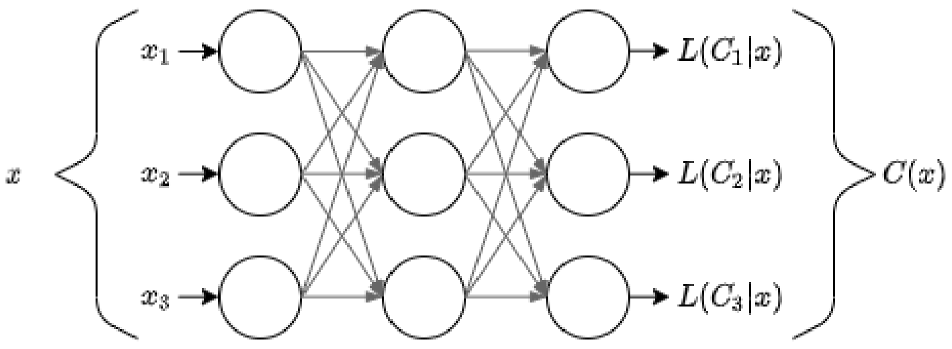

Figure 2.4: Example dense neural network with an input dimension of 3 features, and an output dimension of 3 classes

between 0 and 1. As a result the output will be in the form of a vector of size 3 (e.g. `[0.2, 0.8, 0.3]`). These values are interpreted as some form of the likelihood for each class. When a definitive value is needed the class with the highest value will be taken as the predicted value. When used in training however, all values are used.

Training and Backpropagation

As the size of a network increases, the number of parameters increase exponentially. It is therefore impossible to manually tune all of the weights, and so a way of programmatically reducing the models error is used. Backpropagation is a method used to calculate a gradient that is needed for adaptation of a neural network (Kelley 1960) or the gradient in weight space for a neural network with respect to a loss function.

For a given network we define the following constants:

L The total number of layers in the network

$w^{(i)}$ The weight of the neuron in the i^{th} layer

$a^{(i)}$ The activation/output of the neuron in the i^{th} layer (also defined as $a^{(i)} = \sigma(z^{(i)})$)

$b^{(i)}$ The bias of the neuron in the i^{th} layer

$z^{(i)}$ The output prior to the application of a non-linear function of the neuron in the i^{th} (i.e. $z^{(i)} = \sum_i^n w^{(i)} a^{(i-1)} + b^{(i)}$ where n is the number of input nodes)

σ The sigmoid function $\sigma(x) = \frac{1}{1+e^x}$

y The desired output of the network

C_j The cost function output of the j^{th} output node

The goal of backpropagation is to find how the cost function changes with respect to each parameter of a given node, or $\frac{\partial C_j}{\partial x}$. We are able to calculate this value using the chain rule.

$$\frac{\partial C_j}{\partial x} = \frac{dC_j}{da^L} \cdot \frac{da^L}{dz^L} \cdot \frac{dz^L}{da^{L-1}} \cdot \frac{da^{L-1}}{dz^{L-1}} \cdot \frac{dz^{L-1}}{da^{L-2}} \cdots \frac{da^1}{dz^1} \cdot \frac{\partial z^1}{\partial x} \tag{2.3}$$

As the gradient of the cost function with respect to the output is required, it is sensible to choose a cost function which has a simple derivative. Two common cost functions are mean squared error and categorical crossentropy (Murphy 2012).

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.4}$$

$$\text{CC} = -\sum_{i=1}^{n} y_i \cdot \log \hat{y}_i \tag{2.5}$$

Where n is the number of output dimensions, y_i is the i^{th} predicted output element, and $\hat{y}_i$ is the i^{th} true value output element.

Note that this algorithm only finds the gradient of the loss function for each of the networks parameters. In order to train the network the parameters have to be altered by some function of that gradient. The most common way to do this is **Stochastic Gradient Descent** (SDG). The basic form of this first calculates the gradient of the loss function using backpropagation, then changes the variables of each node from the gradient of the loss function for each variable multiplied by a learning rate. This process is then repeated with the new values for the neural network. This results in the model gradually walking down the gradient until hopefully it finds a point of minimum error, where the model performs as accurately as possible for the training data.

This type of function is referred to as the optimiser function.

2.3.4 Autoencoders

Autoencoders are a type of deep neural network. In a typical neural network the network tries to classify or generate some sort of insight as output from the data provided to it. Autoencoders instead try to replicate the data passed as input as best it can. If the network structure had the same number of nodes at each layer, this wouldn't be particularly useful, it would just generate a one to one mapping. However by altering the number of nodes in each layer, it is able to generate interesting latent features inside the network.

Autoencoders have a bottleneck layer where there are fewer neurons than the other layers. By restricting the number of features available to the network to pass on information, it has to attempt to create latent features at this restrictive layer that encapsulate the most meaning possible.

An autoencoder consists of 2 parts. The first part is the encoder where the dimensionality reduction takes place. The second part is the decoder where the data is reconstructed. These two parts are separated by this bottleneck layer (Liou et al. 2014).

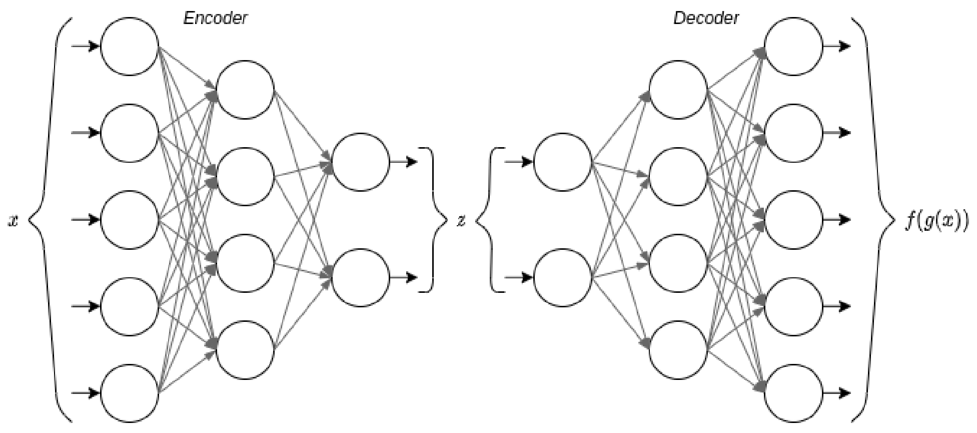

Figure 2.5: Example of an autoencoder with 5 input features and 2 latent features.

An autoencoder is trained in one part, i.e. the encoder and decoder are trained together placed in series. The loss is calculated by comparing some function of the difference between x and $f(g(x))$ where x is a data entity, f is the decoder function, and g is the encoder function (for example the mean squared error). The latent features are also denoted at z but note that $z = g(x)$.

Convolutional Variational Autoencoders

Variational autoencoders apply a probabilistic approach to the autoencoder model. Like a regular autoencoder they are trained to replicate a data point, but at the bottleneck layer instead of just passing the computed latent feature values, a distribution is sampled around the computed point and then passed onto the decoder. The idea behind this is that similar feature values should produce similar results.

As a result of this probabilistic take the loss function is also different. Instead of looking at the difference between 2 specific values, the difference between 2 distributions is used. The evidence lower bound function is used for this (Yang 2017).

$$\log p(x) \geq \mathrm{ELBO} = \mathbb{E}_{q(z|x)}\left[\log \frac{p(x,z)}{q(z|x)}\right] \tag{2.6}$$

Where x is the input value, q is the encoder, p the decoder, and z the latent feature distribution. In order to enter this distribution into the decoder, we perform a Monte Carlo simulation on z and then reconstruct the distribution after it's been passed through the decoder. The Monte Carlo estimate of this expectation is:

$$\approx \log p(x|z) + \log z - \log q(z|x) \tag{2.7}$$

This loss function algorithm was taken from TensorFlow 2021.

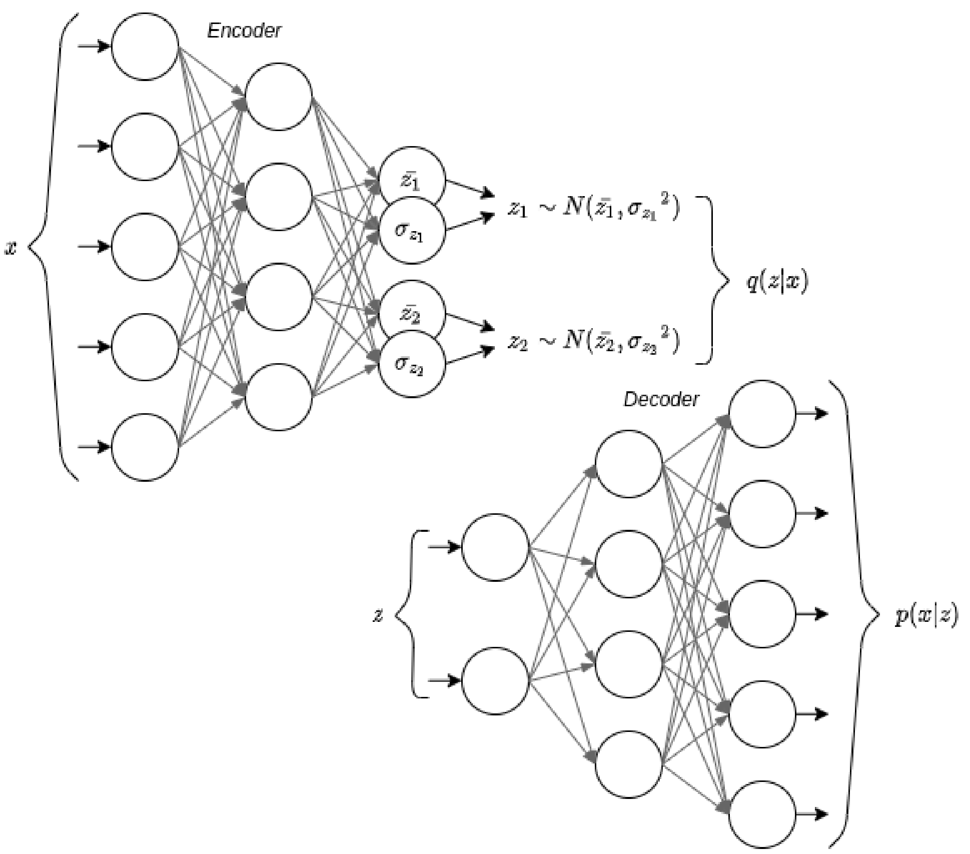

Figure 2.6: Example of a variational autoencoder with 5 input features and 2 latent features.

The number of output notes in the encoder is double that of the decoder. The encoder nodes are paired and one is taken as the mean of a distribution, and the other as the standard deviation. This is referred to as 'reparameterization'. Each feature is constructed with the following form:

$$z = \mu + \sigma \odot \epsilon \tag{2.8}$$

Where ϵ is an error term and distributed as a standard normal distribution.

Also note that:

- The decoder is the same as a regular autoencoder in terms of structure, but the encoder now produces a distribution for each latent feature as an output, rather that a single value.
- When constructing latent features for training purposes only the mean output is used (or rather the error is set to 0).

See the code listing A.1 for an example of this algorithm implemented in Python.

Over/Under-training

Overtraining can occur when too much training is done. As a result of this, the neural network is too closely aligned to the training data, so when new data is introduced, it is unable to generalise and provides poorer results.

Undertraining is when the opposite happens: not enough training has occurred and so when the model looks at new data, the model is not complex enough to classify it correctly.

These two issues can be mitigated by tracking the metrics when training the network. At the point where performance peaks or stagnates is the optimal amount of training. Any more may lead to overtraining, and any less may lead to undertraining.

Local Minima

As mentioned in section 2.3.3, neural networks are trained by gradually descending down the gradient of the loss function. This technique however is not guaranteed to find the best possible parameters for the network, and in certain cases can produce a poor network if the model gets suck in a local minima during training. Figure 2.7 shows an example of a loss function with 2 minima. The first path on the left displays what happens when a training attempt is stuck in a local minima. It is unable to escape from that point, and is therefore unable to train the model as well as possible. The other attempt starting at the center is able to find the global minima, but this is dictated by luck. To try and mitigate this problem, the optimiser function can be altered. A typical way to do this is by adding a momentum parameter into SDG. This adds some function that remembers the previous loss gradient and determines the next update as a combination of the new gradient and this saved previous one.

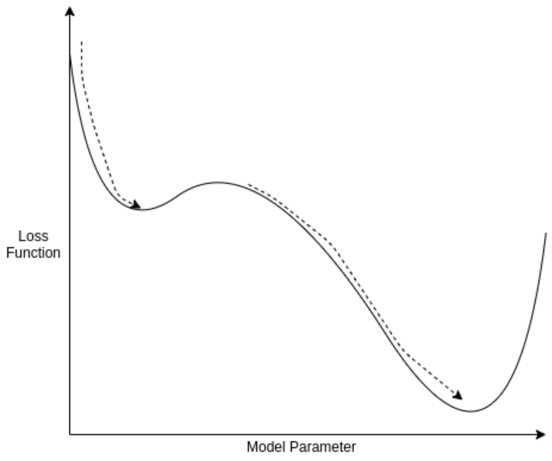

Figure 2.7: Example of a loss function gradient with a local minimum

2.3.5 Support Vector Machines

Support Vector Machines (SVMs) attempt to find the best hyperplane in N-dimensional space that separates distinct classes (Cortes and Vapnik 1995).

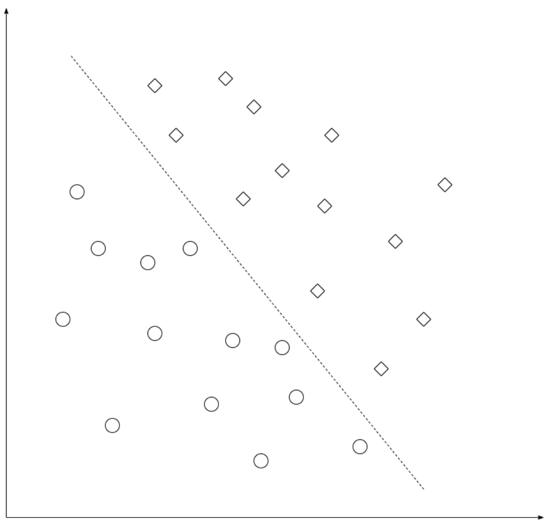

Figure 2.8: 2D Linear Hyperplane separating 2 classes

Figure 2.8 shows an example hyperplane separating 2 classes. However, this plane chosen is one of many possible that would be able to correct discern the correct classes. SVMs provide a method to choose an optimal plane. By minimising the margin between the classes a hyperplane can be found that has the maximum separation.

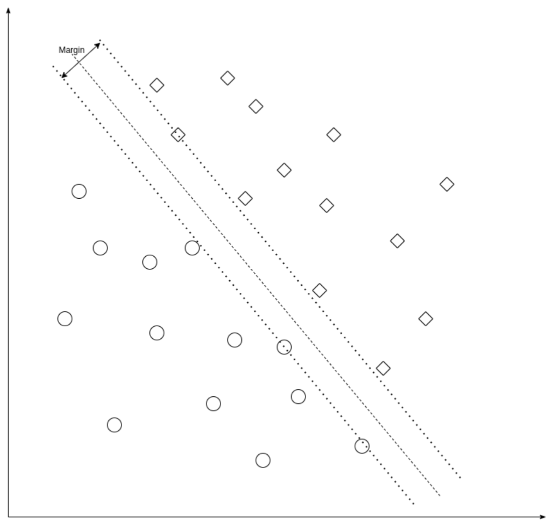

Figure 2.9: 2D Linear Hyperplane margin

This hyperplane can be written as $w^T x - b = 0$ where w is the normal vector to the hyperplane and $\frac{b}{||w||}$ is the offset from the origin to the hyperplane along the normal vector w. In the case where the classes are linearly separable, it is possible to find an exact plane that cuts through the classes without any class impurity on any side (no incorrectly classified samples). In this case the upper margin can be written as $w^T x - b = 1$ where anything above or on this boundary is class "1" and $w^T x - b = -1$ where anything below or on this boundary is class "-1". The distance between these two hyperplanes is $\frac{2}{||w||}$. Hence to minimise the distance between these two hyperplanes $||w||$ needs to be minimised with the condition that there is no class impurity introduced.

In the case that the classes are not linearly separable a slightly different approach needs to be taken. Instead function 2.9 should be minimised.

$$\left[\frac{1}{n}\sum^{n}\max(0, 1 - y_i(w^T x_i - b))\right] + \lambda||w||^2 \tag{2.9}$$

Where n is the number of samples, x_i is the ith sample, y_i is the correct class of x_i, and λ is a constant to denote the weight between margin size, and ensuring that classes are classified correctly.

The term inside the sum is the *hinge loss* function. Whenever a class is correctly classified this function is 0, otherwise is is positive. When this function is summed it provides some metric of total class purity.

Radial Basis Function

Often classes will not be linearly separable to such extent that attempting to fit a line to the samples will prove to be extremely ineffective. A method called the *kernel trick* can be applied to the vector space of the samples which adds extra dimensionality which hopes to create linear bounds between classes.

One method to create this extra dimensionality is using the Radial Basis Function (RBF) (Vert, Tsuda, and Schölkopf 2004):

$$K(x, x') = \exp(-\gamma||x - x'||^2) \tag{2.10}$$

For each sample point, the euclidean distance is calculated between it and every other sample point, and that new dimension is added. This can result in classifiers that are similar to figure 2.10.

2.4 Dimensionality reduction

When performing analysis on data it is often the case that there are too many features within the data to gather any understanding of the patterns within it. Often some of the data's features have very little influence on the category to which a data point belongs (This is in the case of a classification problem). In order to reduce the amount of features a data set has, features can either be removed or combined in order to reduce the number of overall features.

2.4.1 PCA

Principal component analysis is a technique that combines features in order to maximise the variance per feature. The data points are projected onto n-dimensional space. The vector direction of highest variance is then calculated. The direction of this vector then becomes the first new feature for the dataset. The n-dimensional space is then altered so all of the variance in the calculated vector is removed. This is then repeated for the new vector space, and the direction of highest variance is then calculated again and selected as the direction of the 2nd feature. This process is then repeated until

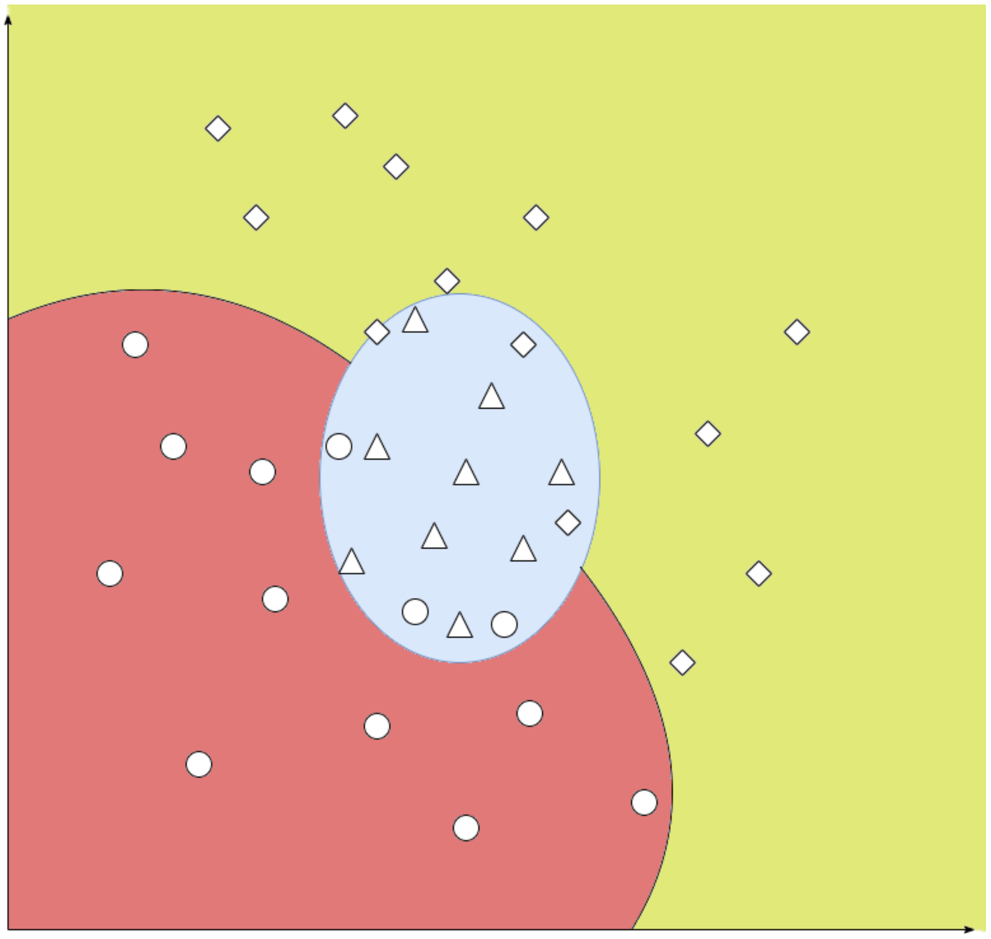

Figure 2.10: Example of a SVM RBF

there are no dimensions left. Note that the number of dimensions created is equal to the original number. This process does not remove or add any data, it only creates new features ordered by variance.

Figure 2.11 demonstrates a PCA transformation on a 2 dimensional dataset. x and y axes are transformed via a change of base into x' and y'. Note that the 2 axes are still perpendicular, so no information is lost or features made dependent on one another.

Variance Threshold

PCA by itself does not reduce the number of features. In order to reduce the number of features a threshold can be set, discarding all remaining features after a certain percentage of variance is obtained. For example, say after PCA has been performed on a dataset, feature 1 has 50% of the variance of the dataset, feature 2 has 26%, and features 3 and 4 have 4% and 2% respectively. With a threshold of 75% features 1 and

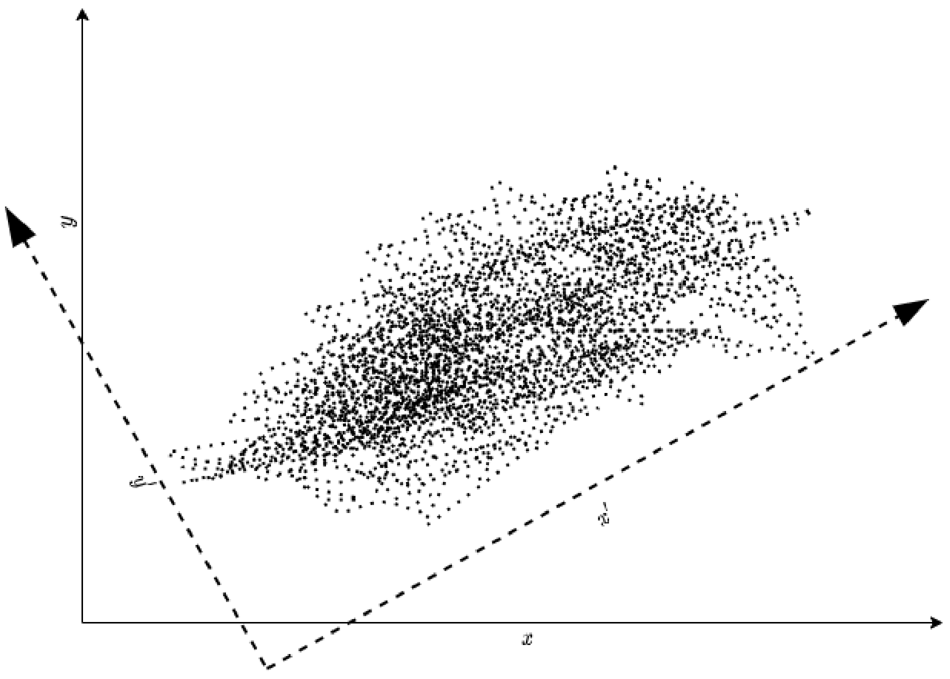

Figure 2.11: Example of a PCA transformation

2 would be retained, and 3 and 4 would be discarded. This percentage can be found by dividing the specific features variance over the total summed variance. Typically this percentage threshold is set to 99% (F.R.S. 1901).

Maximum Likelihood Estimate

With variance thresholding the percentage cutoff is arbitrary. This still is often enough but MLE provides a method attempt to select the optimum number of features. ‘Automatic choice of dimensionality for PCA’(Minka 2000) describes a method using Bayesian model selection to determine the true dimensionality of the data.

2.4.2 Latent Autoencoder Features

Previously, the mechanisms and theory behind autoencoders were discussed. Autoencoders have multiple uses, such as de-noising, anomaly detection, and image generation. This documents discusses and uses them in the context of dimensionality reduction.

As mentioned previously the middle of the network of an autoencoder is purposely restricted to a smaller number of features than the input space. As a results of reducing this feature space the model is likely to produce latent features at this bottleneck which contain the highest amount of 'information'. These latent features can then be used for machine learning, reducing the feature space.

Unlike PCA which is only able to create features which are totally independent of each other, Autoencoders are able to produce features that have some amount of dependency on each other.

2.5 Metrics

In order to evaluate the strength of machine learning models, multiple metrics can be used. This report uses 5 distinct metrics. Each of the following subsection variables are defined as so:

y -set of true class labels

y_{pred} -set of the predicted class labels

n -is the total number of samples

y_i -i^{th} item in the y set

$y_{pred,i}$ -i_{th} item in the y_{pred} set

L -the set containing all samples grouped into sets by class

l -the l^{th} class in the L set

y_l -the set of true class samples of class l

$y_{pred,l}$ -the set of predicted class samples of class l

2.5.1 Accuracy

Accuracy is the percentage of correct classification over the total number of samples. The equation is the same for both binary and multiclass classification.

$$\frac{1}{n}\sum_{i=0}^{n-1} 1(y_{pred,i} = y_i) \tag{2.11}$$

2.5.2 Precision

Precision is the ability of the classifier to label a specific class correctly.

For binary classifications the equation is:

$$\text{precision}(y, y_{pred}) = \frac{|y_{pred,true} \cap y_{true}|}{|y_{true}|} \tag{2.12}$$

For multiclass classifications the weighted precision will be used:

$$\frac{1}{\sum_{l \in L} |y_{pred,l}|} \sum_{l \in L} \frac{|y_{pred,l}||y_{pred,l} \cap y_l|}{|y_l|} \tag{2.13}$$

2.5.3 Recall

Recall is the percentage of components classified correctly for a class, over the total number of items classified as that class.

For binary classifications the equation is:

$$\text{precision}(y, y_{pred}) = \frac{|y_{pred,true} \cap y_{true}|}{|y_{pred,true}|} \tag{2.14}$$

For multiclass classifications the weighted recall will be used:

$$\frac{1}{\sum_{l \in L} |y_{pred,l}|} \sum_{l \in L} \frac{|y_{pred,l}||y_{pred,l} \cap y_l|}{|y_{pred,l}|} \tag{2.15}$$

2.5.4 F1

In terms of precision and recall, the F1 score is defined as:

$$F1 = 2\,\frac{precision \times recall}{precision + recall} \tag{2.16}$$

Note that this contains all the same components as accuracy. However it attempts to form a better overall representation of the performance of a classifier by favouring smaller classes. As accuracy is just an amalgamation of the correctly classified samples over the total number, it favours larger classes. By using the weighted values of precision and recall, F1 score mitigates this problem.

2.5.5 Matthews Correlation Coefficient

From scikit-learn 2021b

Matthews correlation coefficient is defined as:

$$MCC = \frac{c \times s - \sum_k^K p_k \times t_k}{\sqrt{(s^2 - \sum_k^K p_k^2) \times (s^2 - \sum_k^K t_k^2)}} \tag{2.17}$$

where

$t_k = \sum_i^K C_{ik}$ The number of times class k truly occurred

$p_k = \sum_i^K C_{ki}$ The number of times class k was predicted

$c = \sum_k^K C_{kk}$ The total number of samples correctly predicted

$s = \sum_i^K \sum_j^K C_{ij}$ The total number of samples

MCC has a similar purpose to the F1 statistic, as in it tries to form a balanced measure of overall performance without favouring larger classes.

2.6 Related Works

Plenty of research has been done in the field on ML but as the field and the applications of autoencoders are so broad, relatively little work has gone into looking in how they can improve classification algorithms. The paper that most inspired this work was "Autoencoder-based Feature Learning for Cyber Security Applications" (Yousefi-Azar et al. 2017). This paper uses a standard autoencoder to generate latent features and then compares the accuracy of a variety of ML algorithms with and without the latent features. It also uses the NSL-KDD and microsoft malware datasets for experiments. Naive Bayes, K nearest neighbours, SVMs, and gradient boosting were trained on these latent features. With the intrusion detection dataset it was found that autoencoders increased the accuracy of Gaussian Naive Bayes, SVMs, and Xgboost (Gradient boosting) up to between 1.00 and 6.74 percent. With the malware dataset, Gaussian Naive Bayes, K-NN (neural network), and SVMs increased their accuracy between 0.7 and 14.2 percent.

In "Deep Learning Approach Combining Sparse Autoencoder With SVM for Network Intrusion Detection" (Al-Qatf et al. 2018) SVMs are trained on latent features using the NSL-KDD dataset. It aims to create an algorithm that can be quickly trained with new data in order to respond to new network intrusion attacks. By using an autoencoder for feature learning and dimensionality reduction it reduces the amount of training needed by the SVM to achieve good results. In order to gain a better understanding of the performance of this method, previous experiments using the NSL-KDD datasets are collated and presented with the results of this new algorithm. The model has 84.96% accuracy for binary classification and 80.48% accuracy for multiclass classification on the KDDTest+ testing set.

"A Deep Learning Approach for Intrusion Detection Using Recurrent Neural Networks" (Yin et al. 2017) uses NSL-KDD to train and evaluate a recurrent neural network. In Recurrent Neural Networks the hidden layer feeds back into itself. This gives the network the ability to have some notion of order/memory, as previously entered samples will not effect the results of subsequent ones. This may be able to gain some idea of the rate at which packets are being sent (This could be useful in identifying DDoS attacks). It was able to achieve a binary classification accuracy of 83.28% and a multiclass classification accuracy of 81.29% on the KDDTest+ dataset.

"Malware Classification Using Static Analysis Based Features" (Hassen, Carvalho, and Chan 2017) focuses on the extraction method of features from the deconstructed assembly files. It uses opcode n-gram and opcode n-gram with control statement shingling in order to extract features and then trains random forests on the data in order to try and find the most effective method. These methods uses a combination of hashing and grouping in order to create these representations. "n-gram" refers to how many instructions are looked at at a time, and the shingling refers to how many bins these instruction groups are hashed in to. It was found that with random forests, opcode 2-gram was the most accurate method of features extraction for the Microsoft malware dataset. As a note, this report only uses unigram (1-gram) and no shingling to extract the features from the malware dataset.

"Malware Detection using Machine Learning and Deep Learning" (Rathore et al. 2018) uses the Microsoft Malware Classification Dataset as well as benign executables sourced from the GNU `binutils` package to compare the effectiveness of various classifiers with different dimension reduction techniques. It uses variance thresholding and autoencoders for the dimensional reductions, and random forests, and deep neural networks for the classifiers. The aim of this paper was to find the most effective method of malware detection, rather than looking at how the different dimension reduction techniques effect the performance. Using 3-cross fold validation to calculate the accuracy, the maximum metric achieved was 99.78% using variance thresholding and random forest.

As a result of the investigation done looking into these papers, the initial experimental method of this report was improved. It was decided that PCA should be also used in order to gauge how other dimensional reduction methods compare to VAEs. It was also decided that multiple performance metrics should be used instead of only accuracy. Binary and multiclass evaluations should be used for the NSL-KDD dataset.

Chapter 3

Datasets

For the experiment two datasets are used. The NSL-KDD network intrusion dataset (New Brunswick 2021) and the Microsoft malware challenge dataset (Microsoft 2018). Both of these datasets have been widely used to create metrics for ML algorithms (See chapter 2.6) and so the results from other papers will also be displayed in the results section as both a sanity check, and also to gauge the effectiveness of the techniques used within this project.

3.1 NSL-KDD dataset

The NSL-KDD dataset is a subset of the KDD'99 dataset (Iformation and Computer Science 1999). This subset was created to try and solve some of the problems in the original set described in "A Detailed Analysis of the KDD CUP 99 Data Set" (Tavallaee et al. 2009). This dataset is comprised of a selection of datafiles, but in this report only 2 are used: `KDDTrain+.txt` and `KDDTest+.txt`.

Both files have the same CSV format with each line representing a packet. Each packet has various attributes including protocol type, logged in, destination, attempted root login, and number of failed attempts. As well as these attributes each of them have a designated class. This will either be "normal" (i.e. a benign request) or one of several malicious classes (e.g. `rootkit`, `nmap`, `buffer_overflow`...). Table 3.1 displays the counts of each class in the training set, and table 3.2 displays the counts of the test set.

This dataset will be used in two ways. With the meta classes to test the multiclass performance of methods, and with the meta classes aggregated into malicious and benign traffic (i.e. binary classification). Note that some subclasses that are present on

Table 3.1: Frequency counts for NSL-KDD multiclass training dataset

Meta-class	Class	Frequency
Normal	normal	67343
DoS	neptune	41214
	smurf	2646
	back	956
	teardrop	892
	pod	201
	land	18
U2R	buffer_overflow	30
	rootkit	10
	loadmodule	9
	perl	3
R2L	warezclient	890
	guess_passwd	53
	warezmaster	20
	imap	11
	ftp_write	8
	multihop	7
	phf	4
	spy	2
Probe	satan	3633
	ipsweep	3599
	portsweep	2931
	nmap	1493

Table 3.2: Frequency counts for NSL-KDD multiclass test dataset

Meta-class	Class	Frequency
Normal	normal	9711
Dos	neptune	4657
	apache2	737
	processtable	685
	smurf	665
	back	359
	mailbomb	293
	pod	41
	teardrop	12
	land	7
	worm	2
	udpstorm	2
U2R	buffer_overflow	20
	ps	15
	xterm	13
	rootkit	13
	loadmodule	2
	perl	2
	sqlattack	2
R2L	guess_passwd	1231
	warezmaster	944
	snmpguess	331
	snmpgetattack	178
	httptunnel	133
	multihop	18
	named	17
	endmail	14
	xlock	9
	xsnoop	4
	ftp_write	3
	phf	2
	imap	1
Probe	mscan	996
	satan	735
	saint	319
	portsweep	157
	ipsweep	141
	nmap	73

the test set are not present on the training set. This is in order to gauge how well the various classifiers are able to generalise the different types of attack and how they are able to respond to new threats (However this will not be investigated in this paper).

Each packet has 43 features (including its class). `difficulty` is also included in this figure, but will not be used for classification. Of the remaining 41 features, 3 are categorical variables and 38 are numerical.

3.2 Microsoft Malware Classification Challenge

The Microsoft Malware Classification Challenge (Microsoft 2018) is a collection of disassembled x86 malware files hosted on Kaggle[1]. This was originally constructed for WWW 2015 (international World Wide Web conference) [2]. It provides, two folders labeled `test` and `train` containing assembly and hex files of malware, `trainLabels.csv` containing classifications of all the files in the `test` folder, and `sampleSubmission.csv` containing a sample submission for the malware challenge. Note that it does not provide a list of classes for the test files. The original aim of this challenge was to identify the most effective method of classifying malware, and as a result participants were not able to see the true classes of the samples they were trying to classify.

For this report it is essential the models generated are evaluated. To do this, data used for training cannot also be used for testing. This ensures that over-fitting does not take place (i.e. where the model is too focused on the training set, and as a result when used with in production/different data, performs poorly). To avoid this, 20% of the training data is set aside and only used for testing/generating metrics. From this, 8695 samples are contained in the training set and 2174 sample are in the testing set.

This report will be using only the assembly files. Each of these files contains a IDA [3] disassembled x86 binary malware file. Each line consists of the type of hex line (i.e. if it's data or code), the instruction offset, the function hex code, and then a list of arguments passed to that code. 2 hex digits are designated as this code, and hence there are 256 possible codes, although not all of these are used.

The distribution of the classes of the entire dataset is displayed in table 3.3.

[1]`https://www.kaggle.com/`

[2]`https://web.archive.org/web/20210117190951/http://www.www2015.it/welcome-letter/`

[3]`https://hex-rays.com/blog/ida-7-6-released/`

Table 3.3: Frequency counts for Microsoft Malware Challenge Dataset

Class	Frequency
Kelihos_ver3	2942
Lollipop	2478
Ramnit	154
Obfuscator.ACY	122
Gatak	101
Tracur	75
Vundo	47
Kelihos_ver1	39
Simda	4

Chapter 4

Experimental Methodology

The goal of this experiment is to identify how altering the features of samples affects the performance of the machine learning classifiers. To do this the experiment can be broken down into several parts.

1. **Dataset Preprocessing** In this step the raw data is transformed into some form that can be used in the machine learning algorithms.
2. **Dimensionality Reduction** This step alters the preprocessed data and performs various types of dimension reduction techniques.
3. **Model Creation/Training** ML models are trained with the provided datasets.
4. **Model Evaluation** The various metrics are now calculated with the models and the testing data.

A diagram of this workflow is presented in figure 4.1.

4.1 Dataset Preprocessing

4.1.1 NSL-KDD

The properties of the NSL-KDD dataset are described in section 3.1. In order for the dataset to be ready for dimensionality reduction, several transformations need to be applied.

Min Max Scaling

The autoencoder network is set so it only takes values between 0 and 1. In order to make the dataset compatible, the numerical features need to be transformed into that range. For each features, the following transformation will be applied:

```python
std = nsldf_train[k].std()
if std != 0:
    minn = min(nsldf_train[k].min(), nsldf_test[k].min())
    maxn = max(nsldf_train[k].max(), nsldf_test[k].max())
    output_train[k] = (nsldf_train[k] - minn)/(maxn - minn)
    output_test[k] = (nsldf_test[k] - minn)/(maxn - minn)
```

Listing 4.1: Min Max Scaling in Python

Where `nsldf_train` is the raw training dataset, `nsldf_test` is the raw testing dataset, `k` is the name of the current numerical feature, and `output_train` and `output_test` are the respective output datasets.

This code scales the values linearly between 0 and 1 from the maximum values to the minimum values of the test and train datasets.

Note that the standard deviation is calculated and the feature is discarded if it is equal to 0. This will only happen when all the values on a features are the same (hence the feature is useless). This also ensures that we do not divide by 0 (as `maxn - minx = 0` in that case).

One-Hot Encoding

Neural networks are unable to deal with categorical variables, and as a results they need to be transformed into some sort of numerical data in order to be processed. For ordinal values this could be done with a ranking system, but the features in this dataset are all orderless, hence compressing one into a single feature would introduce extra information into the data, where there should be none.

One-hot encoding avoids this problem by splitting up a single categorical variable into multiple features. For each distinct category, a new feature is made. For each entity that is a member of that category, the value of that feature is set to 1, otherwise it is set to 0. The result of this encoding may look like in table 4.1.

The exception to this transformation is the class feature, which is handled on a model by model basis.

Table 4.1: Example of one-hot encoding

Name	animal_cat	animal_dog	animal_moose
Scooby Doo	0	1	0
Top	1	0	0
Bullwinkle	0	0	1

4.1.2 Microsoft Malware Classification Challenge

The Microsoft Malware Dataset consists of a number of x86 assembly files (ASM) and a CSV file detailing the type of malware each ASM file is (More information of this set is detailed in section 3.2). Each of the lines in a ASM file has a number of hex digits. The first 2 begin the instruction to be executed on the set, and the rest are the arguments. This report will use a similar method as the winning entry of the Microsoft Malware Classification Challenge (Liu 2015) to manipulate the data into a machine learning processable form: a unigram representation.

To form a unigram representation, the frequency of each hex instruction set is found, and then put through a log transformation. As the instruction set is 2 hex digits, we know there are only 256 instruction, and hence 256 features are generated. The equation of frequency to unigram is:

$$\text{unigram}(f) = \begin{cases} 0, & \text{if} f = 0 \\ 1 + \log f, & \text{otherwise} \end{cases} \tag{4.1}$$

After this, min max scaling is applied to the dataset as described previously.

4.2 Dimensionality Reduction

At this stage the chosen dimensional reduction methods are applied. There are specific parameters peculiar to each of these methods. By varying these parameters a better understanding of the effectiveness of each method can be achieved. Each of the following methods are applied to the raw datasets, and then passed on.

None

To gather baseline metrics it is essential that we pass on the data unmodified. This gives a comparison to see the effectiveness of the other dimensional reduction techniques.

PCA Variance Threshold

PCA is applied to the dataset, and then a percentage variance threshold is applied. This threshold is varied between 90% and 99%.

PCA MLE Threshold

PCA is applied to the dataset, and then a MLE threshold is applied.

Variational Autoencoder

A variational encoder is trained on the dataset, and then from that the latent features are extracted. The number of epochs are varied from 100 to 500, and the number of latent features are also varied from 2 to 14.

Combinations

Combinations of PCA and autoencoders are also constructed. For each PCA type, a set of autoencoders are trained with the same parameter variances for each.

4.3 Model Creation/Training

For each of the different versions of dimensionality reduction, several models are trained.

Gaussian Naïve Bayesian Networks

There are no parameter for this model.

SVM

A RBF kernel is used for the SVM

Decision Trees

For dtrees, the following parameters are varied.

- `criterion` : `gini` or `entropy`
- `splitter` : `best` or `random`

- `min_samples_split` : 2, 4, 6, or 8
- `min_samples_leaf` : between 1 and 4 inclusive

4.4 Model Evaluation

At this stage each model is possessed with the appropriate test data into the metric generation. The following metrics are then generated:

- Accuracy
- Precision
- Recall
- F1
- Matthews Correlation Coefficient

These are described in section 2.5.

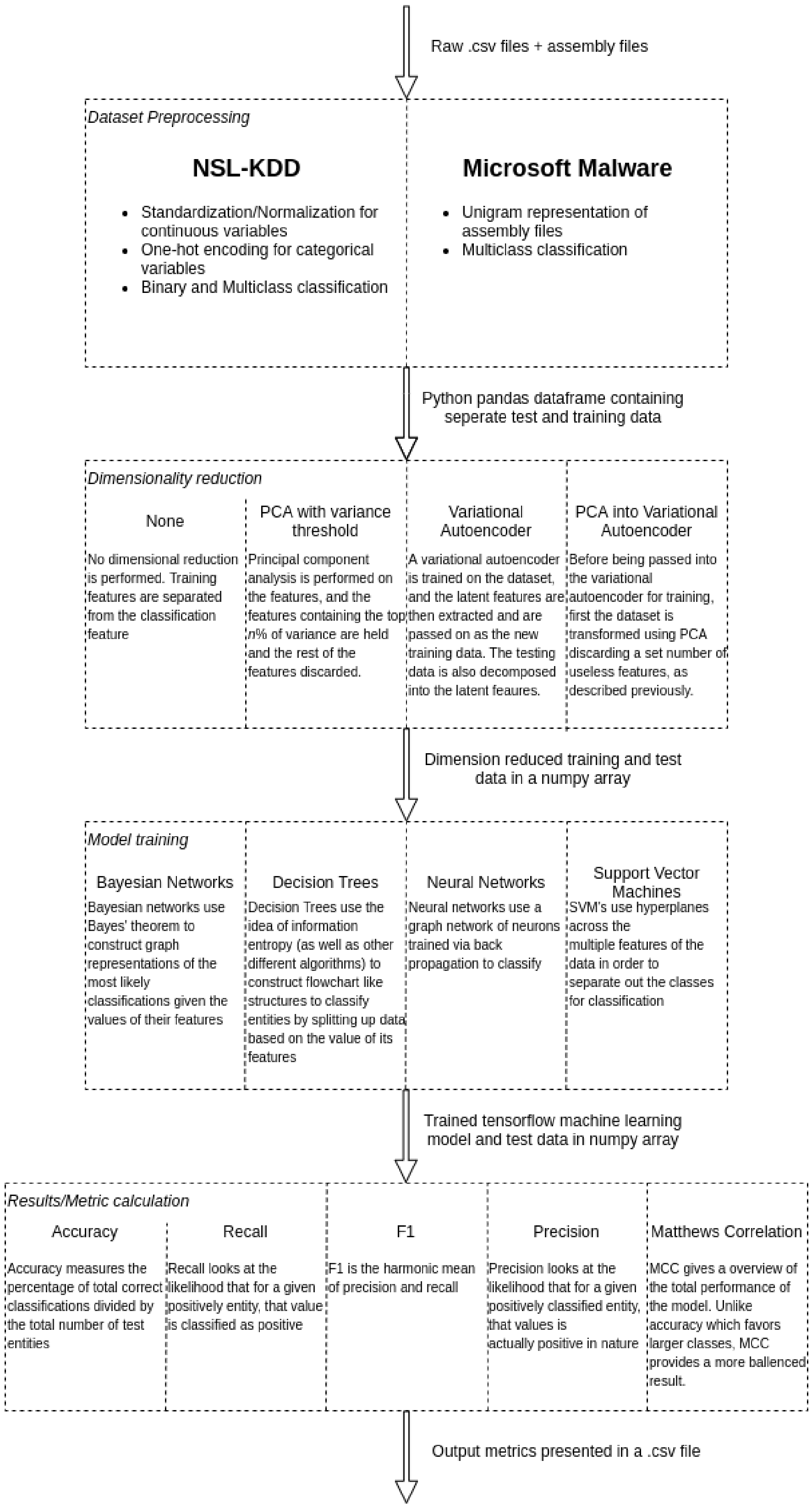
Raw .csv files + assembly files
Dataset Preprocessing
NSL-KDD
Standardization/Normalization for continuous variables
One-hot encoding for categorical variables
Binary and Multiclass classification
Microsoft Malware
Unigram representation of assembly files
Multiclass classification
Python pandas dataframe containing seperate test and training data
Dimensionality reduction
None
No dimensional reduction is performed. Training features are separated from the classification feature
PCA with variance threshold
Principal component analysis is performed on the features, and the features containing the top n% of variance are held and the rest of the features discarded.
Variational Autoencoder
A variational autoencoder is trained on the dataset, and the latent features are then extracted and are passed on as the new training data. The testing data is also decomposed into the latent feaures.
PCA into Variational Autoencoder
Before being passed into the variational autoencoder for training, first the dataset is transformed using PCA discarding a set number of useless features, as described previously.
Dimension reduced training and test data in a numpy array
Model training
Bayesian Networks
Bayesian networks use Bayes' theorem to construct graph representations of the most likely classifications given the values of their features
Decision Trees
Decision Trees use the idea of information entropy (as well as other different algorithms) to construct flowchart like structures to classify entities by splitting up data based on the value of its features
Neural Networks
Neural networks use a graph network of neurons trained via back propagation to classify
Support Vector Machines
SVM's use hyperplanes across the multiple features of the data in order to separate out the classes for classification
Trained tensorflow machine learning model and test data in numpy array
Results/Metric calculation
Accuracy
Accuracy measures the percentage of total correct classifications divided by the total number of test entities
Recall
Recall looks at the likelihood that for a given positively entity, that value is classified as positive
F1
F1 is the harmonic mean of precision and recall
Precision
Precision looks at the likelihood that for a given positively classified entity, that values is actually positive in nature
Matthews Correlation
MCC gives a overview of the total performance of the model. Unlike accuracy which favors larger classes, MCC provides a more ballenced result.
Output metrics presented in a .csv file

Figure 4.1: Diagram of data flow and software architecture

Chapter 5

Experimental Implementation

5.1 Framework Breakdown

To implement this experiment `Python` was chosen to construct the framework for testing due to the large data science community that surrounds it. Due to the number of variables in this experiment it was necessary that the experiment could be broken up into sections, so multiple computers could run it at the same time. Two meta stages are used as a result of this need: Autoencoder training and saving, and model creation and metric generation.

Autoencoder training and saving In this step all variational autoencoders are trained on the relevant data and then saved using `TensorFlows` inbuilt serialization. Once this step is completed, inside `models/`, there are three folders: `mal/`, `nsl_bin/`, and `nsl_multi/`. They each contain all the various autoencoder configurations trained on them.

Model creation and metric generation In this step, for each encoder trained the relevant dataset is transformed into its latent features representation. Then the various machine learning models are trained on those new data representations. Once trained, the predicted values for the test sets are then compared with the true values, and the relevant metrics are generated.

5.2 Technologies

Below the lists how each of the following technologies are implemented.

Preprocessing To store and manipulate the raw data `pandas` and `numpy` are used.

Variational Autoencoder `TensorFlow` is used to construct the variational autoencoders, using 2 sequential networks trained in series.

PCA `scikit-learn` is used to perform PCA on the datasets. Including its implementation of MLE.

Gaussian Naïve Bayesian Networks `scikit-learn`s implementation of Gaussian Naïve Bayesian networks are used.

Decision Trees `scikit-learn`s implementation of decision trees are used.

Support Vector Machines `scikit-learn`s implementation of SVMs are used, specifically using the RBF kernel.

Dense Neural Networks `TensorFlow` is used to create the dense neural networks, using the sequential model, categorical crossentropy as the loss function, and the 'adam'optimizer for learning.

Metric Calculation `scikit-learn.metrics` package is used for all metric calculation. Accuracy is calculated using `accuracy_score`, precision with `precision_score`, recall with `recall_score`, F1 with `f1_score` and MMC with `matthews_corrcoef`. For multiclass metrics it is also necessary for the precision, recall, and F1 score to be supplied with an '`average`'argument. This is needed as those three metrics are not inherently multiclass, so in order for them to be useful for multiclass representations they need to be adapted. This report chose to use the `'weighted'` option for the average argument as well as setting `zero_division=0` for the case where it is attempted to divide by 0.

Visualisations To create the visualisations in chapter 6 `matplotlib`, `seaborn`, and `jupyter` are used.

5.3 Neural Network Architecture

When referring to autoencoders or dense neural networks the structure of them is defined generally, but not specifically. This means that certain decisions need to be made to create the exact structure of each network.

For the variational autoencoders, the input size is set by the number of features the training data has, and the number of latent features is set as one of the variable which is altered within the experiment. The hidden layers however are not defined. For this experiment the layers are defined as:

- Encoder layer 1 (input layer): size of `feature_dim`
- Encoder layer 2 (hidden layer): size of `int((feature_dim + latent_dim) / 2)`
- Encoder layer 3 (output layer): size of `latent_dim * 2`
- Decoder layer 1 (input layer): size of `latent_dim`
- Decoder layer 2 (hidden layer): size of `int((feature_dim + latent_dim) / 2)`
- Decoder layer 3 (output layer): size of `feature_dim`

Where `feature_dim` in the integer value of the number of features of the training data, `latent_dim` is the number of latent features chosen for this specific autoencoder, and `int((feature_dim + latent_dim) / 2)` is the floored average of the 2 dimensions. Note how the encoder has double the output dimensions. This is discussed in section 2.3.4.

Each VAE was trained for 200 epochs and used a 'Adams'optimiser with a learning rate of 0.0001.

Similarly, for the dense neural network a similar decision has to be made. The number of input nodes is equal to the number of dataset features, the number of output nodes is defined as the number of potential classes, and the number of hidden layers is defined as a variable inside the experiment, but the number of nodes inside each hidden layer is not. It is arbitrarily defined as so:

```python
def lin_layers(indim, outdim, n_layers, n):
  grad = (outdim - indim) / n_layers
  return int(indim + (n + 1) * grad)
```

Listing 5.1: Node density function

This gradually transitions from the number of input dimensions to the number off output dimensions (specifically from number of features to number of possible classes). The number of training epochs is set as a constant 20. This value was based off light testing, coming to the conclusion that at around 20 epochs the loss function tended to stabilise.

For NSL multiclass and malware multiclass the ‘Adams’optimiser was used to train the network with all default parameters. For NSL binary stochastic gradient decent was used with a learning rate of 0.02 and momentum set to 0.5. This was to try and reduce the number of models that were stuck in a local minimum. This was believed to be caused by the smaller model size due to the reduced number of output nodes.

5.4 Hardware

This program was run on a number of machines. The machines and the programs run on them are listed in table 5.1.

Table 5.1: Table describing the machines used to run this experiment and the parts of the experiment run on each of them

OS	CPU	GPU	Programs
Arch Linux x86_64	AMD Ryzen 3600	NVIDIA GeForce GTX 1050	Malware Autoencoder Training Malware ML Model Training Malware Metric Generation NSL Binary Autoencoder Training NSL Binary ML Model Training NSL Binary Metric Generation
Debian 5.10.46-4 x86_64	Intel i5-4440	N/A	NSL Multiclass Autoencoder Training NSL Multiclass ML Model Training NSL Multiclass Metric Generation
Manjaro Linux x86_64	Intel i5-5300U	N/A	Visualisation Generation

Chapter 6

Experimental results

6.1 Multiclass Malware Evaluation

This section will look at the performance of the machine learning classifiers for the multiclass malware dataset.

6.1.1 Dimensional Reduction Tuning

Variational Autoencoders

Figure 6.1 shows how average performance across all models and latent feature dimensions increases as the model is trained. On average the most accurate models are trained to the maximum epoch of 500. This suggests that more training was needed for the autoencoder to have maximum effect. However, the graphs show a trend of the metrics leveling out so not much more training should be required to reach an optimal model. For analysing the performance of the ML classifiers we shall only look at the autoencoders trained to 500 epochs.

Figure 6.2 shows how varying the number of latent feature dimensions effects the overall performance average across all models. The maximum average performance is at 14 latent feature dimensions. The shape of the graphs rapidly increases at lower latent dimensions, before mellowing out but still increasing slightly. It would be expected if an optimal number of latent dimensions had been reached for performance to decrease. This suggests that the optimal number of latent features for this dataset is in fact more that 14.

Figure 6.3 displays the average effect on performance when including a PCA threshold before the data is passed into an autoencoder. The overall maximum performance

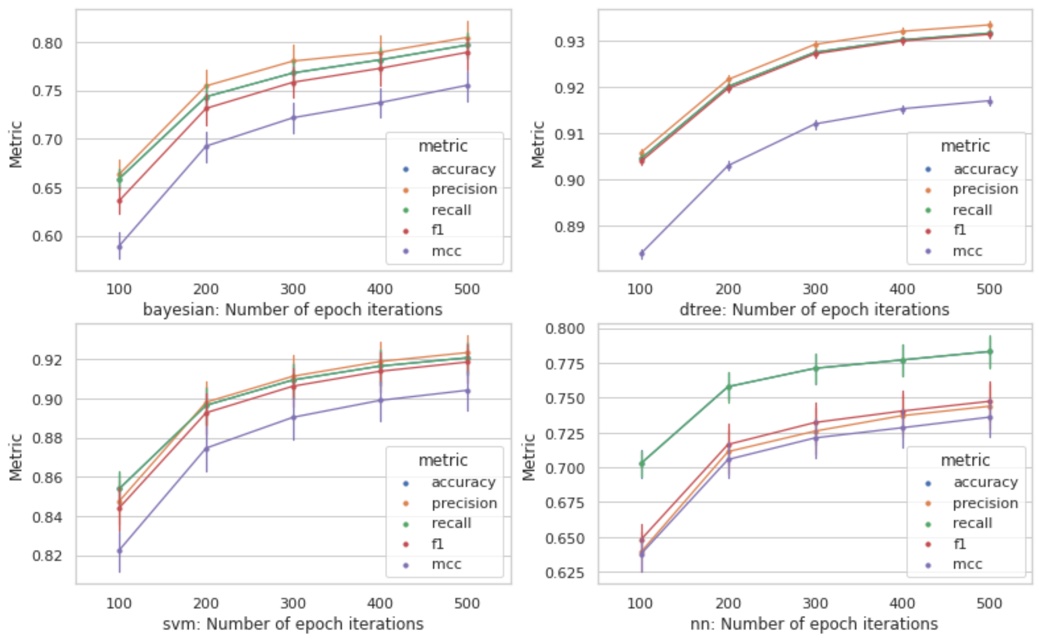

Figure 6.1: Malware multiclass: Average performance across all combinations with an autoencoder against number of epochs trained

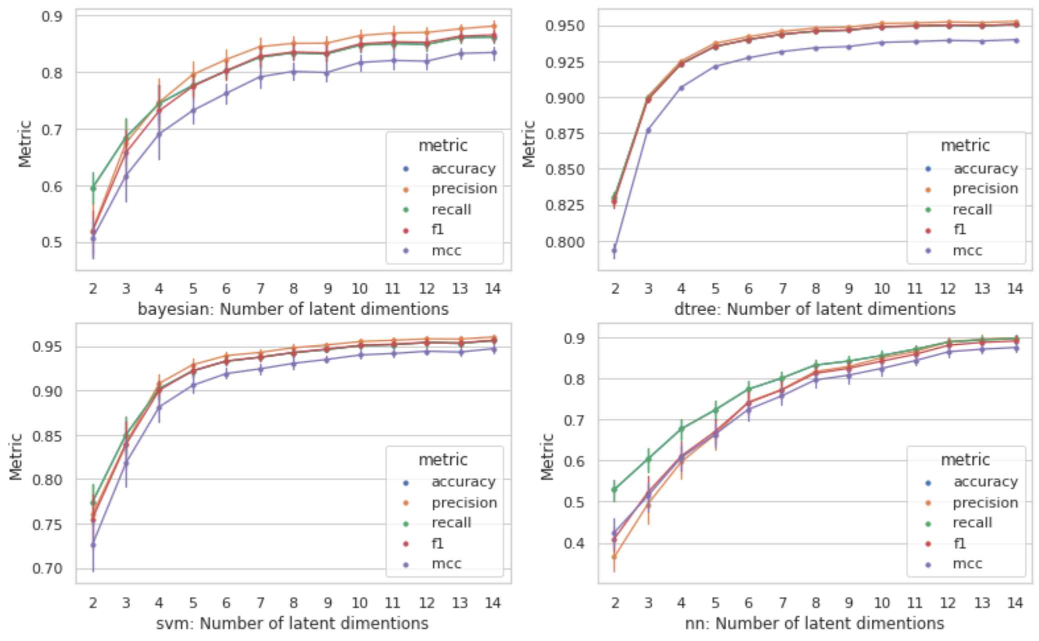

Figure 6.2: Malware multiclass: Average performance across all combinations with an autoencoder against number of latent feature dimensions

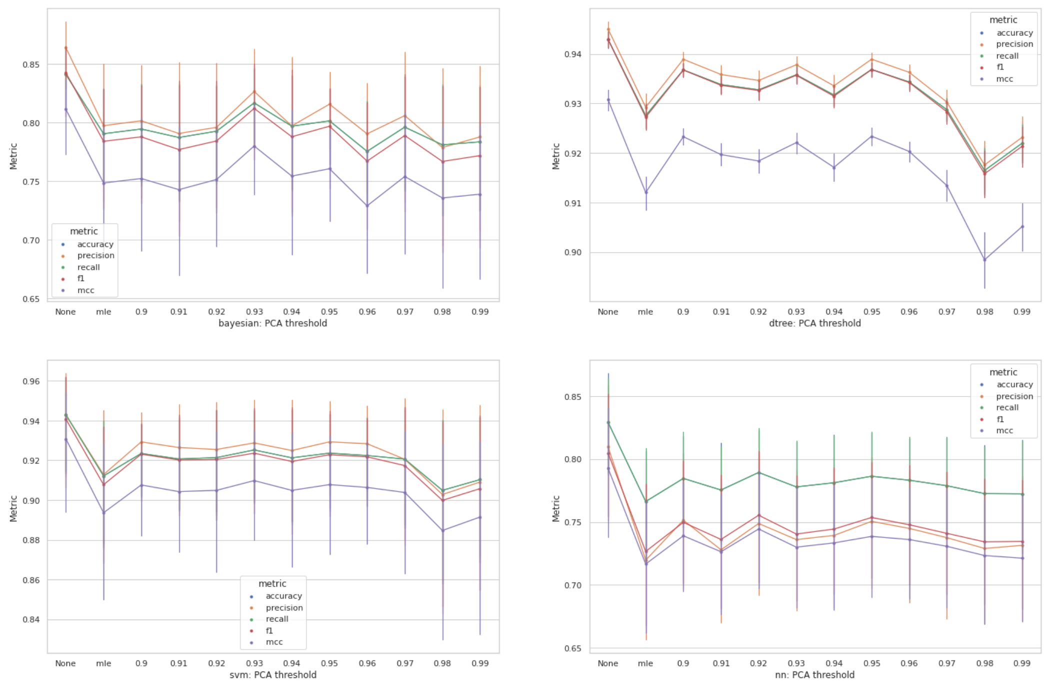

Figure 6.3: Malware multiclass: Average performance across all combinations with an autoencoder against type of PCA variance threshold

happens when PCA is not applied at all ($y = none$).

PCA Variance Threshold

Figure 6.4 displays the average effect on performance when including a PCA threshold before using ML classifiers. On average having a higher threshold seems to increase the performance. However, as the variance is so massive it is impossible to draw any meaningful conclusion.

6.1.2 Classification Evaluations

For each model type, this subsection will cover the effect dimensionality reduction has on the performance of them.

Gaussian Naïve Bayesian

Figure 6.5 shows the effect of the three different types of dimensional reduction on each of the performance metrics. The line in blue is the metric showing the performance of the model without any DR techniques applied.

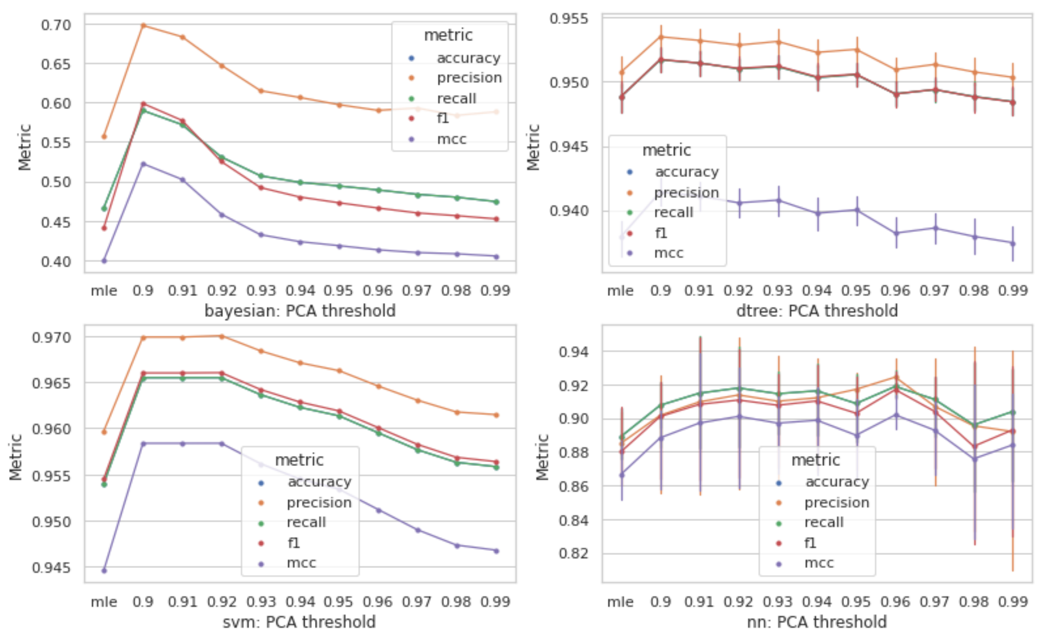

Figure 6.4: Malware multiclass: Average performance across all combinations of ML classifier against type of PCA variance threshold

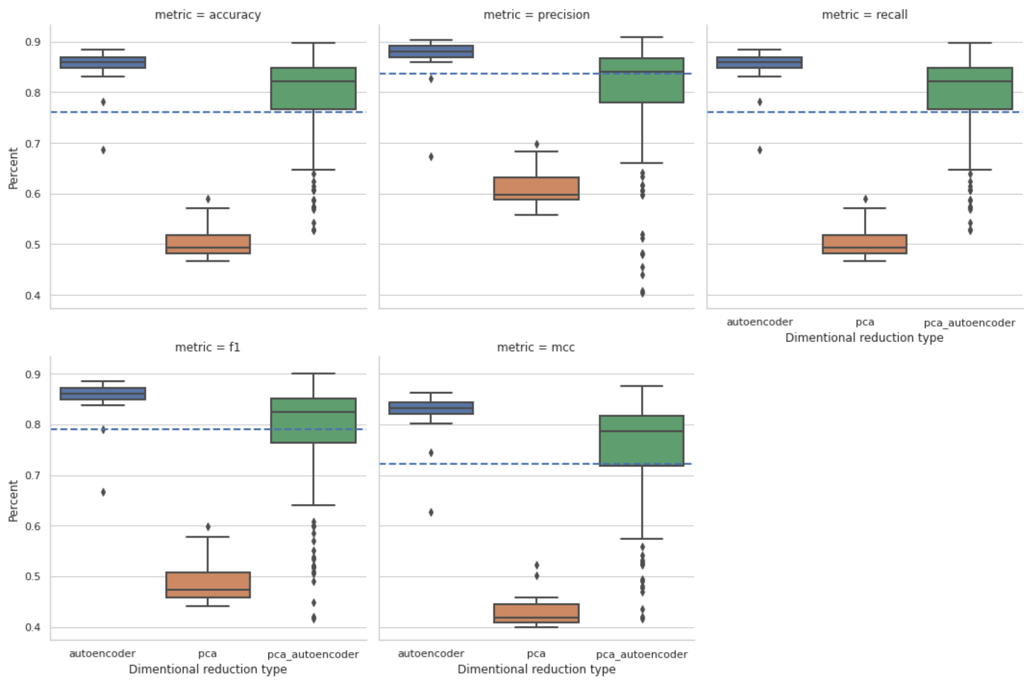

Figure 6.5: Malware multiclass: Metric boxplot for all types of metric for Gaussian Naïve Bayes split by dimensional reduction method

Table 6.1: Malware multiclass: Mean improvement for each dimensional reduction technique from base metric for Gaussian Naïve Bayesian Networks

DR type	VAE	PCA	PCA + VAE
Accuracy	0.080709	-0.252697	0.032150
Precision	0.028050	-0.221489	-0.036818
Recall	0.080709	-0.252697	0.032150
F1	0.052196	-0.297502	-0.006111
MCC	0.089619	-0.286058	0.028179

Table 6.2: Malware multiclass: Maximum improvement for each dimensional reduction technique from base metric for Gaussian Naïve Bayesian Networks

Meta DR type	DR	Accuracy	Precision	Recall	F1	MCC
None	none	0.760350	0.835816	0.760350	0.790289	0.721719
VAE	500 epochs 9 features	0.883625	0.903141	0.883625	0.885346	0.861733
PCA	thresh 0.9	0.589696	0.697568	0.589696	0.598669	0.522298
PCA w VAE	thresh 0.93 & 500 epochs 11 features	0.896964	0.908418	0.896964	0.899499	0.875812

This figure shows that on average the most effective method of increasing the performance of Bayesian networks is using a variational autoencoder. Table 6.1 displays the mean increase in performance for each metric and DR technique. Note that for each metric, VAEs increase the performance.

The dimensional reduction technique that produced the highest metrics for Naïve Gaussian Bayes' classifier was a variational autoencoder with PCA trained for 500 epochs, with 9 latent feature dimensions and a threshold of 0.93. Table 6.2 shows the maximum performing dimensional reduction techniques for each type of DR.

Support Vector Machine with RBF Kernel

Figure 6.6 displays each of the various metrics for a SVM and the effect on them from altering the DR method. None of the DR methods increased the performance of the SVM. Table 6.3 shows the average difference for each meta type of DR.6.4 show the maximum performing DR technique by MCC for each meta DR type. Note that 'none'has the highest performance with an accuracy of 0.978381 and a MCC of 0.973872. This is the highest for all of the techniques used in this dataset.

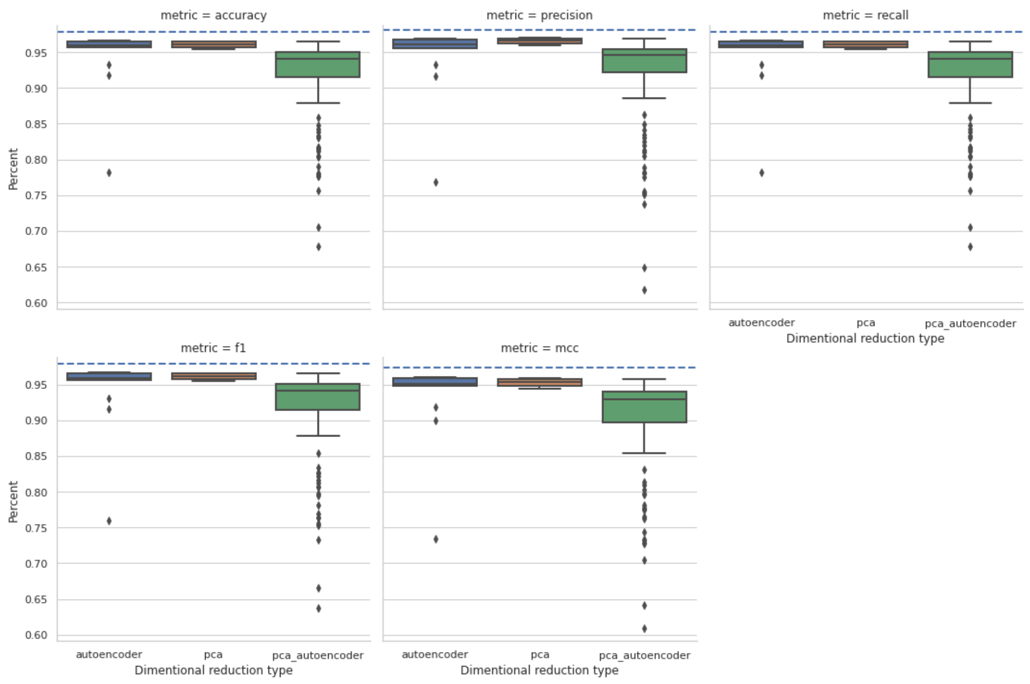

Figure 6.6: Malware multiclass: Metric boxplot for all types of metric for SVM RBF kernel split by dimensional reduction method

Table 6.3: Malware multiclass: Mean improvement for each dimensional reduction technique from base metric for SVMs with RBF kernel

DR type	VAE	PCA	PCA + VAE
Accuracy	-0.035454	-0.017730	-0.059643
Precision	-0.038088	-0.015336	-0.059375
Recall	-0.035454	-0.017730	-0.059643
F1	-0.037845	-0.017392	-0.062034
MCC	-0.043085	-0.021328	-0.072141

Table 6.4: Malware multiclass: Maximum improvement for each dimensional reduction technique from base metric for SVMs with RBF kernel

Meta DR type	DR	Accuracy	Precision	Recall	F1	MCC
None	none	0.978381	0.981003	0.978381	0.978586	0.973872
VAE	500 epochs 14 features	0.966881	0.969140	0.966881	0.967059	0.959881
PCA	threshold 0.92	0.965501	0.970070	0.965501	0.966044	0.958378
PCA w VAE	threshold 0.94 & 500 epochs 14 features	0.964581	0.968419	0.964581	0.965034	0.957207

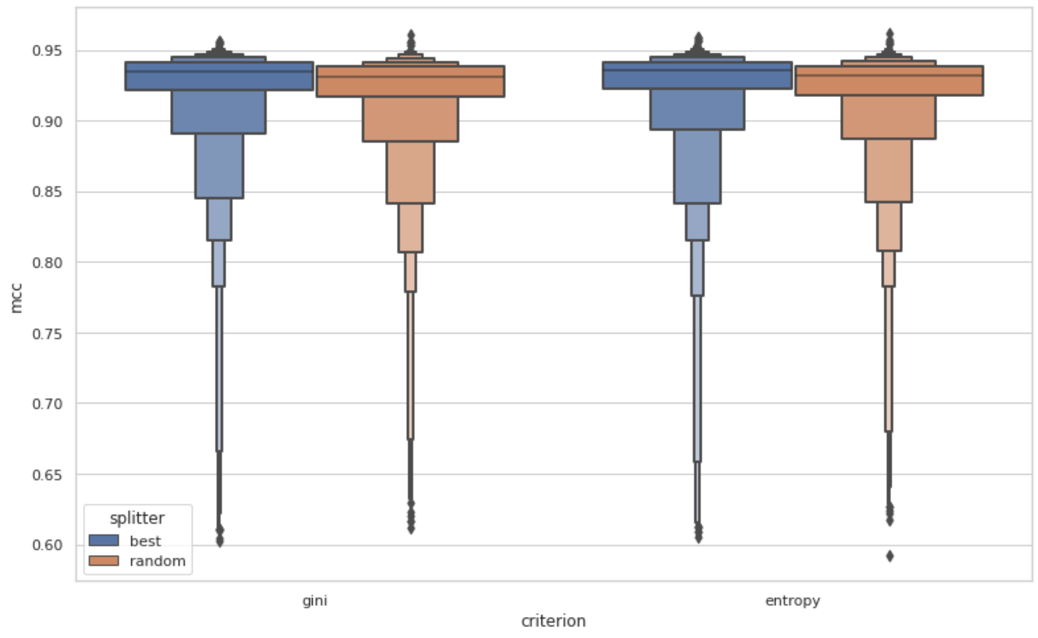

Figure 6.7: Malware multiclass: Boxen plot of decision tree MCC split by classification criterion and node splitting strategy

Table 6.5: Malware multiclass: Mean improvement for each dimensional reduction technique from base metric for Decision Trees

Criterion	Splitter	MSS	MSL	DR type	Accuracy	Precision	Recall	F1	MCC
gini	best	2	1	none	0.964121	0.966050	0.964121	0.964319	0.956490
gini	random	2	1	none	0.967341	0.968906	0.967341	0.967405	0.960417
entropy	best	6	3	none	0.965961	0.969278	0.965961	0.966430	0.958822
entropy	random	6	1	none	0.968261	0.970090	0.968261	0.968447	0.961506

Decision Trees

As mentioned in section 4.3 there are 4 parameters that are varied with the decision tree model. 6.7 shows the effect on MCC of altering the `criterion` and `splitter` variables. On average `best` splitting (MCC mean of 0.909629 for `entropy` and 0.907961 `gini`) performs better than `random` (MCC mean of 0.905033 for `entropy` and 0.904319 `gini`). However, the maximum performing models for both `gini` and `entropy` criterion are `random`. Table 6.5 lists the maximum values for each split.

To compare the effectiveness of the DR techniques as accurately as possible, it is necessary that when observing their effectiveness we only change the DR technique, and not the model. With this set of decision trees there are 246 unique models each with different parameters. Instead of life in previous sections where we have created

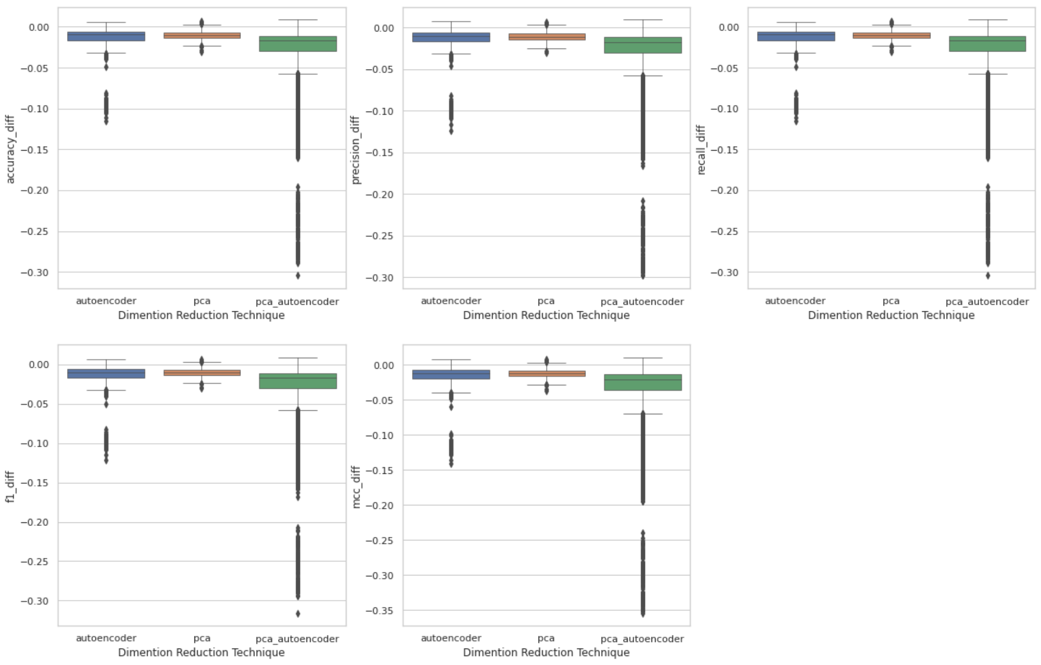

Figure 6.8: Malware multiclass: Decision tree metric improvement box plots for each metric split by DR technique

a set of box plots of each metric and specified the base metric, table 6.8 displays the distribution of the differences for each metric, combining each of the different model types into this visualisation.

The average improvement is negative for each DR technique, but each of the different classes have cases where the DR technique improves the performance of the model. The highest performing model with an accuracy of 0.968261 and a MCC of 0.961506 does not use any form of dimensional reduction, uses `entropy` criterion and `random` splitting, and has a MSS of 6 and a MSL of 1.

Neural Networks

Figure 6.9 shows the average performance of the dense neural network over the depth of the hidden layers without any DR techniques. The maximum number of layers 5 performs the best overall which is to be expected (mean of 0.974701 accuracy and 0.969373 MCC). Also on average 3 layers perform worse than 1, but only by 0.00046 for accuracy and 0.000547 for MCC.

Figure 6.10 shows distribution box plots for each of the metrics differences for each model. It shows that none of the DR methods increase the performance of the models. The maximum performing model is the 5 layer neural network with no DR. It has an

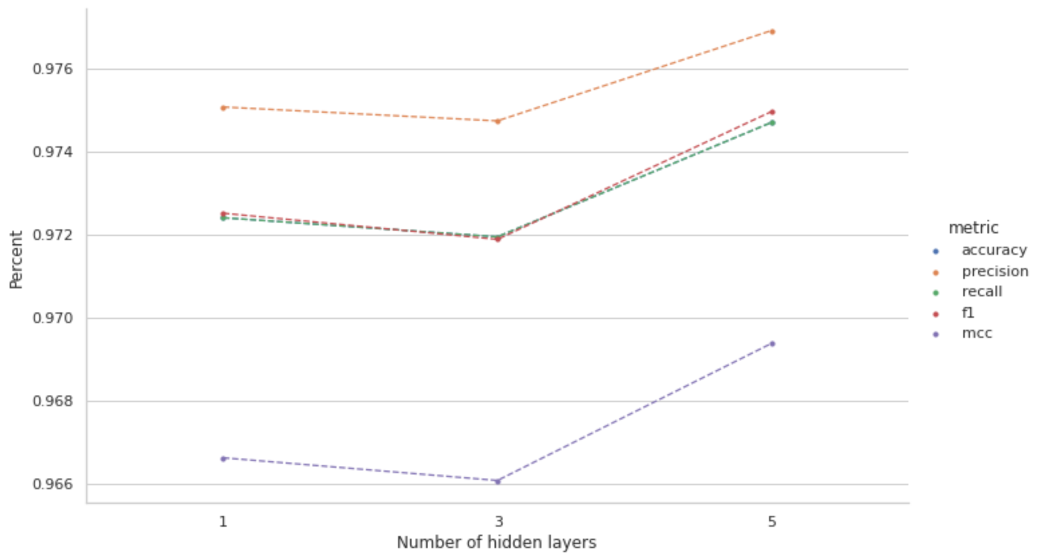

Figure 6.9: Malware multiclass: Model performance over model hidden layer depth without DR

accuracy of 0.974701 and a MCC of 0.969373.

Performance

Table 6.6 shows the top 10 performing models and the associated DR technique.

Table 6.6: Malware multiclass: Top 10 performing models

DR type	Model type	Accuracy	Precision	Recall	F1	MCC
none	dtree entropy best mss 2 msl 2	0.965961	0.968943	0.965961	0.966470	0.958795
VAE 500 14	svm rbf	0.966881	0.969140	0.966881	0.967059	0.959881
VAE 500 13	svm rbf	0.966881	0.968566	0.966881	0.967123	0.959862
VAE 500 11	svm rbf	0.966881	0.969088	0.966881	0.965648	0.959864
none	dtree gini random mss 2 msl 1	0.967341	0.968906	0.967341	0.967405	0.960417
none	dtree entropy random mss 6 msl 1	0.968261	0.970090	0.968261	0.968447	0.961506
none	nn 3	0.971941	0.974737	0.971941	0.971880	0.966065
none	nn 1	0.972401	0.975071	0.972401	0.972510	0.966612
none	nn 5	0.974701	0.976916	0.974701	0.974963	0.969373
none	svm rbf	0.978381	0.981003	0.978381	0.978586	0.973872

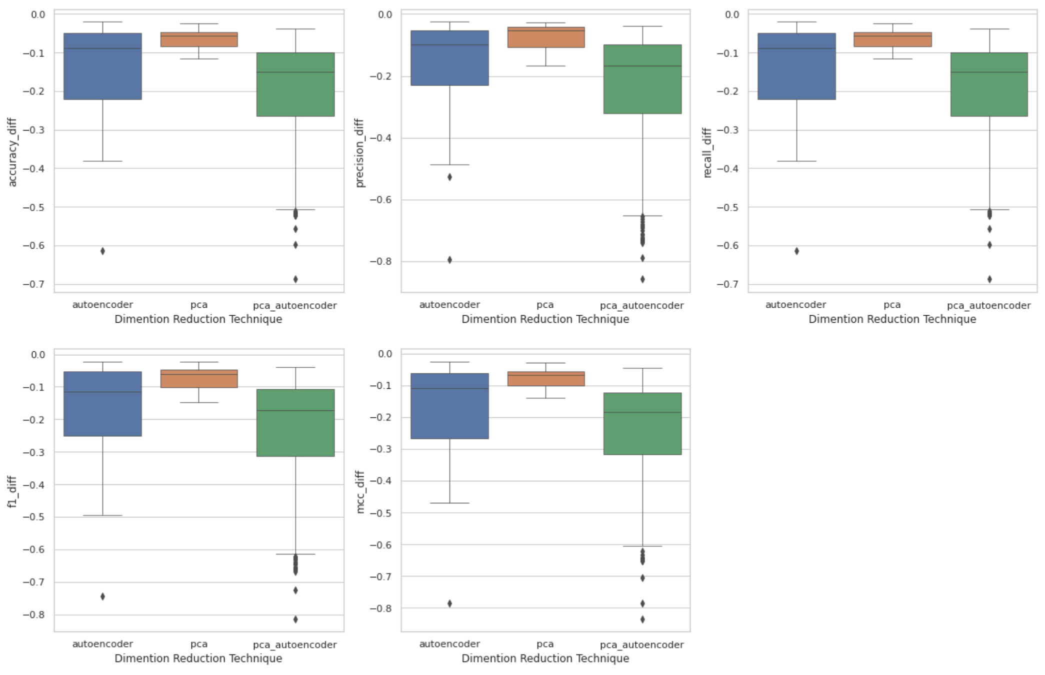

Figure 6.10: Malware multiclass: Box plot of each metric improvement for neural networks split by DR technique

6.2 NSL-KDD Binary Classification

This section looks the performance of the ML classifiers for the binary version of the NSL-KDD dataset

6.2.1 Dimensional Reduction Tuning

Variational Autoencoders

Figure 6.11 shows how average performance across all models and latent feature dimensions changes as the model is trained. From the graph it appears that the autoencoder is already mostly trained at 100 epochs. However at 500 epochs the autoencoders still have the best average performance with a mean accuracy of 0.700356 and a mean MCC of 0.451333 overall. As a result, only the 500 epoch autoencoders results will be analysed for the rest of this section.

Figure 6.12 shows how varying the number of latent feature dimensions effects the overall performance average across all models. In general as the number of latent dimensions increase, overall so does the performance of the ML classifiers, however this relationship is very loose, and so no latent dimension results will be discarded for

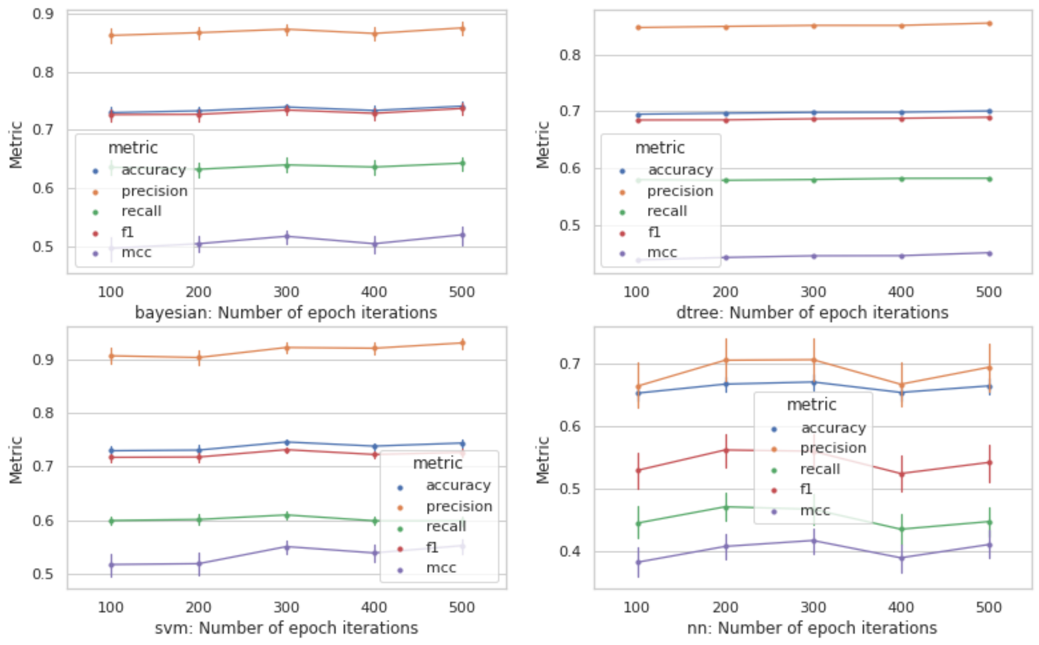

Figure 6.11: NSL-KDD Binary: Average performance across all combinations with an autoencoder against number of epochs trained

the next part of the analysis.

Figure 6.13 displays the average effect on performance of including a PCA threshold before the data is passed into an autoencoder. This graph shows that performance definitely increases on average when some form of PCA thresholding is used. However, it is inconclusive if there is some optimal amount.

PCA Variance Threshold

Figure 6.14 displays the average effect on performance when including a PCA threshold before using ML classifiers. On average having a higher threshold seems to increase the performance. However, as the variance is so massive it is impossible to draw any meaningful conclusion.

6.2.2 Classification Evaluations

For each model type, this subsection will cover the effect dimensionality reduction has on the performance of them.

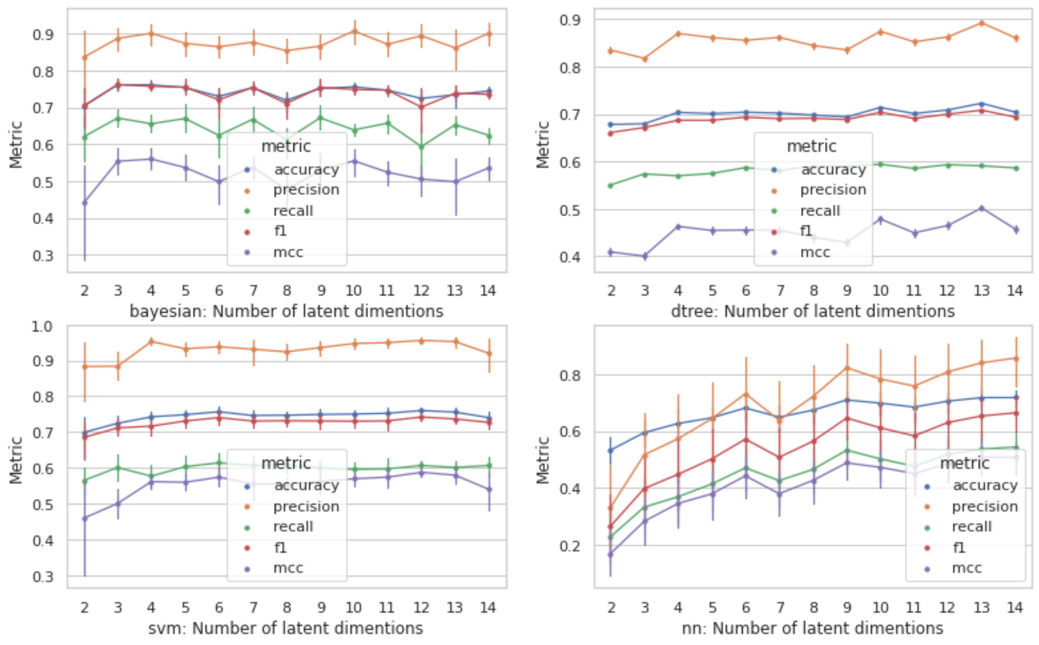

Figure 6.12: NSL-KDD Binary: Average performance across all combinations with an autoencoder against number of latent feature dimensions

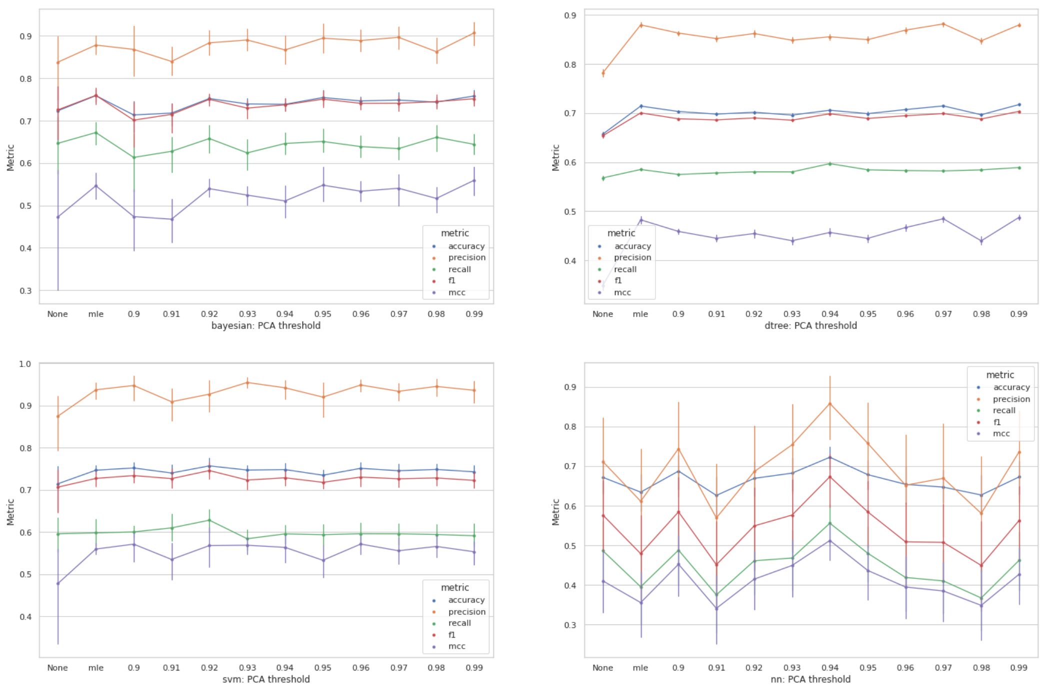

Figure 6.13: NSL-KDD Binary: Average performance across all combinations with an autoencoder against type of PCA variance threshold

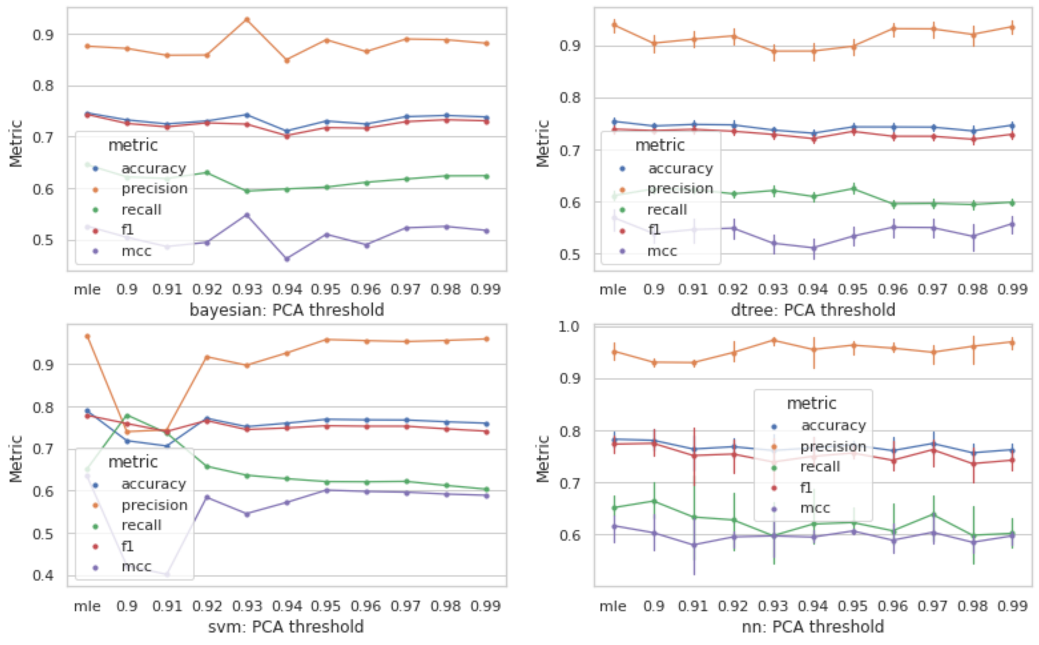

Figure 6.14: NSL-KDD Binary: Average performance across all combinations of ML classifier against type of PCA variance threshold

Table 6.7: NSL-KDD Binary: Mean improvement for each dimensional reduction technique from base metric for Gaussian Naïve Bayesian Networks

DR type	VAE	PCA	PCA + VAE
Accuracy	0.271496	0.280635	0.290982
Precisioon	0.324512	0.364020	0.366039
Recall	-0.065930	-0.095875	-0.069866
F1	0.128701	0.127246	0.141484
MCC	0.692453	0.726935	0.743009

Gaussian Naïve Bayesian

Figure 6.15 shows the effect of the three different types of dimensional reduction on each of the performance metrics. The line in blue is the metric showing the performance of the model without any DR techniques applied.

This figure shows that each of the three DR methods produce some improvement in all of the metrics, apart from recall. However, the two statistics that are constructed from precision and recall (namely F1 and MCC) still increase so the net gain of performance is still positive. Table 6.7 displays the mean increase in performance for each metric and DR technique.

The dimensional reduction technique that produced the highest overall metrics is

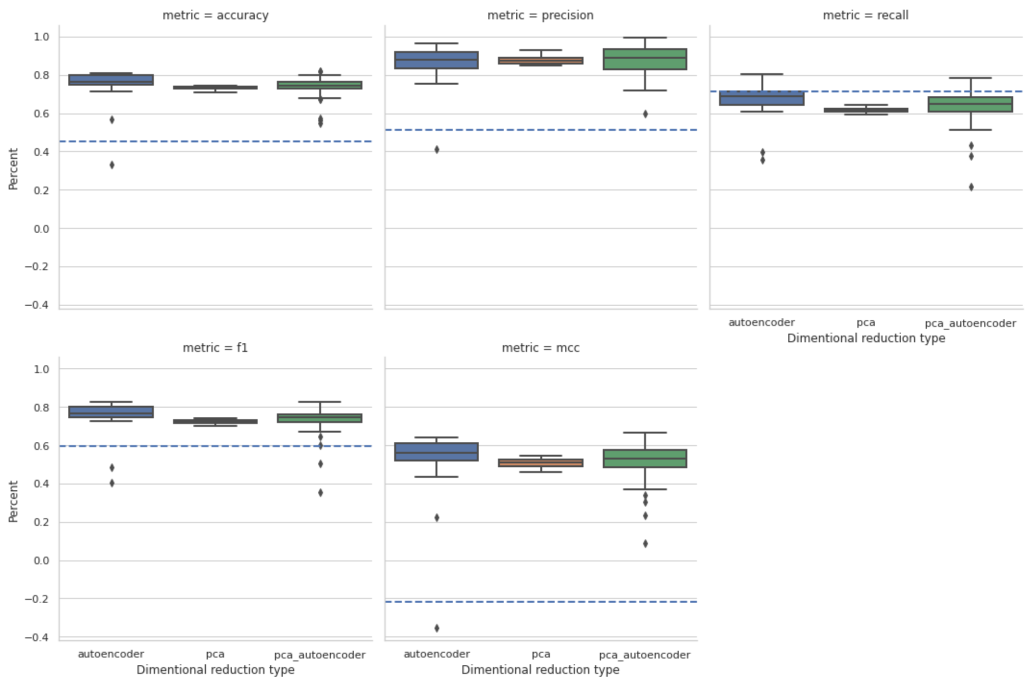

Figure 6.15: NSL-KDD Binary: Metric boxplot for all types of metric for Gaussian Naïve Bayes split by dimensional reduction method

Table 6.8: NSL-KDD Binary: Maximum improvement for each dimensional reduction technique from base metric for Gaussian Naïve Bayesian Networks

Meta DR type	DR	Accuracy	Precision	Recall	F1	MCC
None	none	0.451694	0.513249	0.712382	0.596639	-0.219382
VAE	400 epochs 5 features	0.810282	0.853437	0.804956	0.828488	0.617966
PCA	MLE threshold	0.745697	0.875423	0.645056	0.742788	0.525569
PCA w VAE	0.97 threshold & 500 epochs 4 features	0.821505	0.937258	0.735681	0.824326	0.667928

a variational autoencoder with a PCA threshold trained for 500 epochs, with 4 latent feature dimensions, and with a threshold of 0.97. Table 6.8 shows the maximum performing dimensional reduction techniques for each type of DR.

Support Vector Machine with RBF Kernel

Figure 6.16 displays each of the various metrics for a SVM and the effect on them from altering the DR method. There are instances where each of the DR methods increased and decreased the performance of the ML models. Table 6.9 shows the average difference for each meta type of DR.6.10 show the maximum performing DR technique for each meta DR type. Note that the ‘PCA w VAE’has the highest performance with

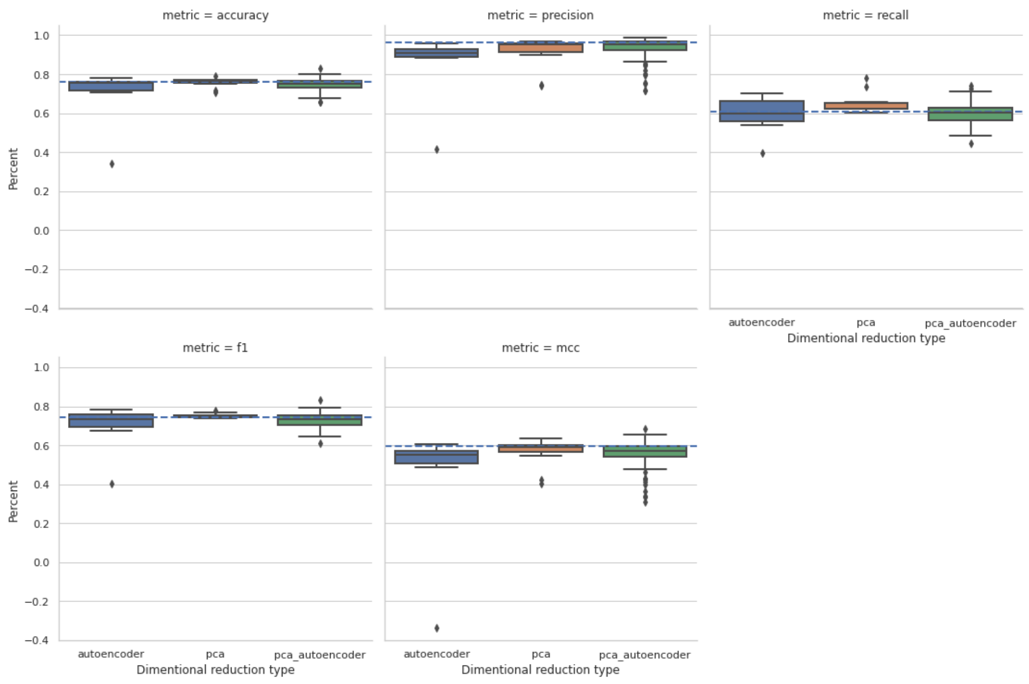

Figure 6.16: NSL-KDD Binary: Metric boxplot for all types of metric for SVM RBF kernel split by dimensional reduction method

Table 6.9: NSL-KDD Binary: Mean improvement for each dimensional reduction technique from base metric for SVMs with RBF kernel

DR type	VAE	PCA	PCA + VAE
Accuracy	-0.048422	-0.005253	-0.016317
Precision	-0.090248	-0.053932	-0.028641
Recall	-0.009585	0.043702	-0.006658
F1	-0.037584	0.009035	-0.015862
MCC	-0.117931	-0.034707	-0.037293

an accuracy of 0.828070 and a MCC of 0.682102. This is the highest for all of the techniques used in this dataset.

Decision Trees

As mentioned in section 4.3 there are 4 parameters that are varied with the decision tree model. 6.17 shows the effect on MCC of altering the `criterion` and `splitter` variables. There is very little different in the means for all four of the splits. Table 6.11 lists the maximum values for each split, but yet again there is very little difference.

To compare the effectiveness of the DR techniques as accurately as possible, it is necessary that when observing their effectiveness we only change the DR technique,

Table 6.10: NSL-KDD Binary: Maximum improvement for each dimensional reduction technique from base metric for SVMs with RBF kernel

Meta DR type	DR	Accuracy	Precision	Recall	F1	MCC
None	none	0.762952	0.965213	0.605392	0.744086	0.595881
VAE	500 epochs 4 features	0.779853	0.933752	0.660095	0.773431	0.604059
PCA	MLE threshold	0.789257	0.968576	0.650900	0.778580	0.634754
PCA w VAE	threshold 0.92 & 500 epochs 6 features	0.828070	0.946555	0.739733	0.830461	0.682102

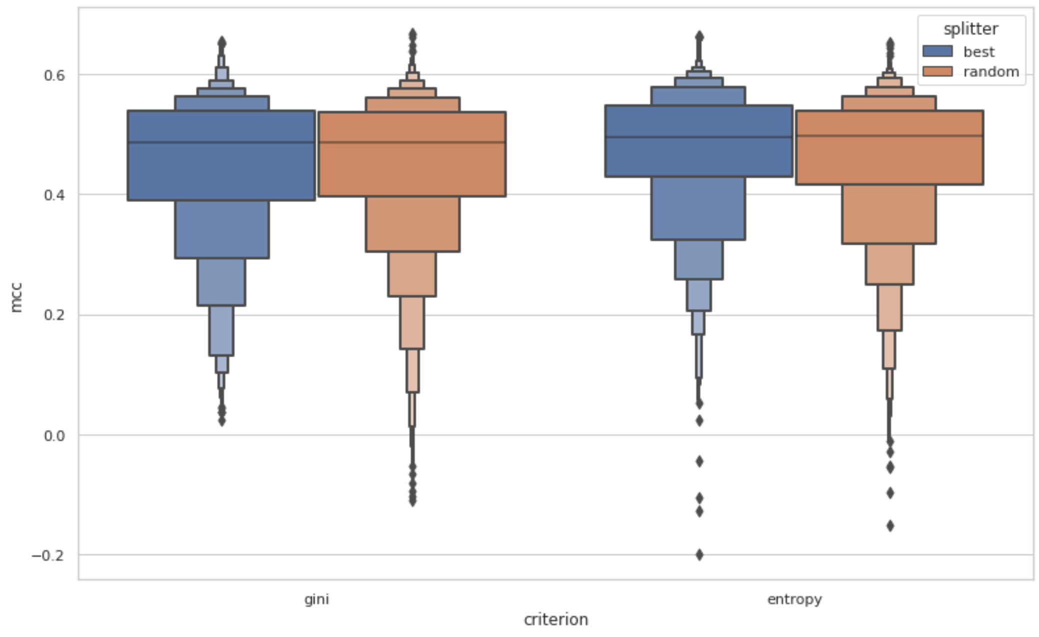

Figure 6.17: NSL-KDD Binary: Boxen plot of decision tree MCC split by classification criterion and node splitting strategy

Table 6.11: NSL-KDD Binary: Mean improvement for each dimensional reduction technique from base metric for Decision Trees

Criterion	Splitter	MSS	MSL	DR type	Accuracy	Precision	Recall	F1	MCC
gini	best	8	3	none	0.801677	0.972750	0.670381	0.793745	0.654712
gini	random	6	3	none	0.811081	0.970169	0.689317	0.805977	0.667231
entropy	best	6	2	none	0.809351	0.967057	0.688537	0.804370	0.663211
entropy	random	2	2	none	0.801411	0.968281	0.673186	0.794208	0.652169

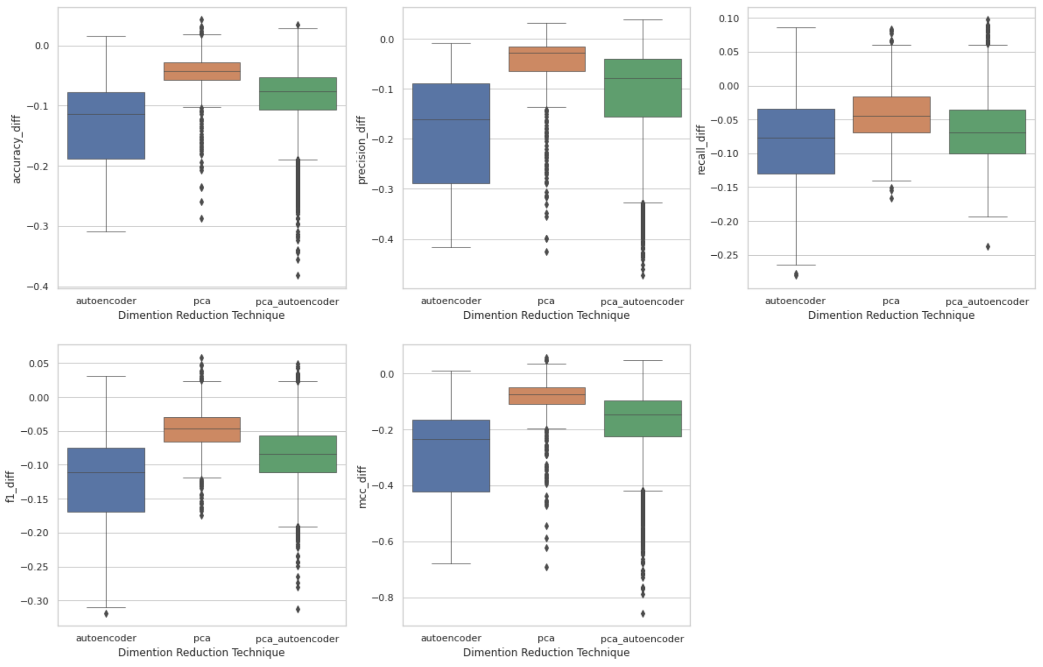

Figure 6.18: NSL-KDD Binary: Decision tree metric improvement box plots for each metric split by DR technique

and not the model. With this set of decision trees there are 246 unique models each with different parameters. Instead of life in previous sections where we have created a set of box plots of each metric and specified the base metric, table 6.18 displays the distribution of the differences for each metric, combining each of the different model types into this visualisation.

The average improvement is negative for each DR technique, but each of the different classes have cases where the DR technique improves the performance of the model. The highest performing model with an accuracy of 0.881081 and a MCC of 0.667231 does not use any form of dimensional reduction, uses `entropy` criterion and `random` splitting, and has a MSS of 6 and a MSL of 3.

Neural Networks

Figure 6.19 shows the performance of the dense neural network over the depth of the hidden layers without any DR techniques. On average it appears that only using 1 layer performs the best in terms of accuracy and MCC with values of 0.734045 and 0.530839 respectively.

Figure 6.20 shows distribution box plots for each of the metrics differences for each model. It shows that on average using any 3 of the techniques reduces the performance

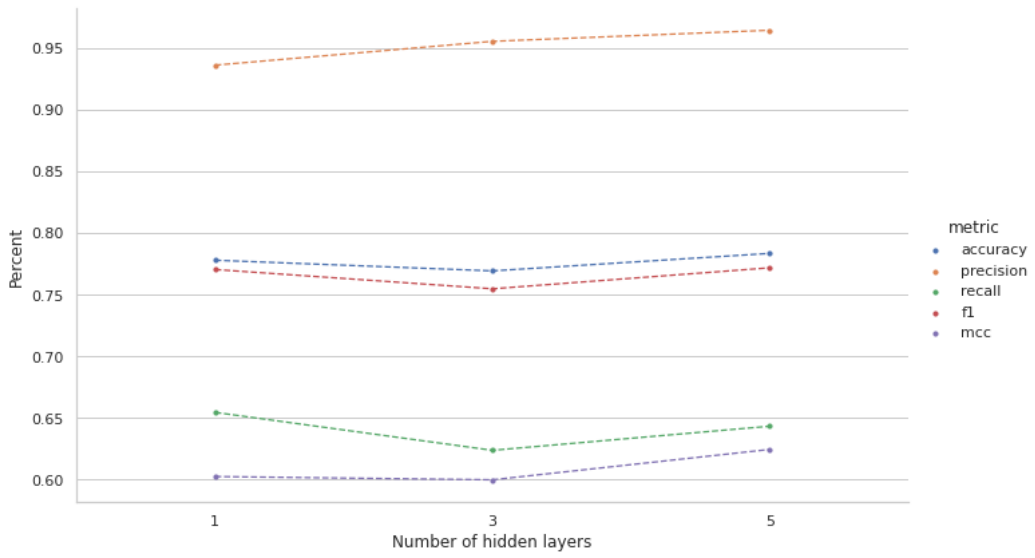

Figure 6.19: NSL-KDD Binary: Model performance over model hidden layer depth without DR

of the NNs, but there are cases where a DR technique increases performance. The neural network with the highest MCC and accuracy was a 3 layered model using both PCA and a VAE with a threshold of 0.9 and trained with 500 epochs and 9 latent features.

Performance

Table 6.12 shows the top 10 performing models and the associated DR technique.

6.3 NSL-KDD Multiclass Classification

This section looks the performance of the ML classifiers for the multiclass version of the NSL-KDD dataset

6.3.1 Dimensional Reduction Tuning

Variational Autoencoders

Figure 6.21 shows how average performance across all models and latent feature dimensions changes as the model is trained. From each of the graphs it appears that the autoencoder is already mostly trained at 100 epochs as there is very little difference between the metrics. For this section all the VAEs of different epochs will be analysed.

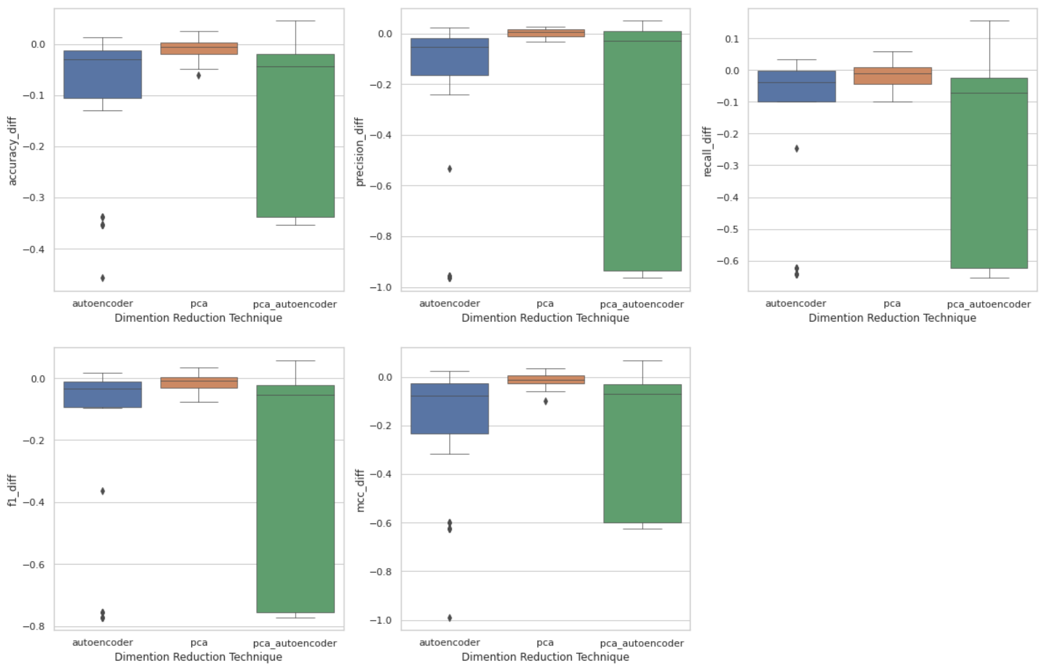

Figure 6.20: NSL-KDD Binary: Box plot of each metric improvement for neural networks split by DR technique

Table 6.12: NSL-KDD Binary: Top 10 performing models

DR type	Model type	Accuracy	Precision	Recall	F1	MCC
none	dtree entropy best mss 8 msl 1	0.806911	0.969027	0.682615	0.800988	0.660562
none	dtree gini random mss 2 msl 2	0.808419	0.966879	0.686979	0.803244	0.661755
none	dtree entropy best mss 4 msl 2	0.808818	0.968554	0.686433	0.803448	0.663137
none	dtree entropy best mss 6 msl 2	0.809351	0.967057	0.688537	0.804370	0.663211
none	dtree gini random mss 4 msl 1	0.809972	0.969880	0.687524	0.804651	0.665462
PCA @ 0.91 VAE 500 ep 11 LFD	nn 1	0.809839	0.971842	0.685810	0.804148	0.666197
none	dtree gini random mss 6 msl 3	0.811081	0.970169	0.689317	0.805977	0.667231
PCA @ 0.9 VAE 500 ep 9 LFD	NN 3	0.814674	0.959541	0.704122	0.812225	0.667584
PCA @ 0.97 VAE 500 ep 4 LFD	bayesian	0.821505	0.937258	0.735681	0.824326	0.667928
PCA @ 0.92 VAE 500 ep 6 LFD	svm rbf	0.828070	0.946555	0.739733	0.830461	0.682102

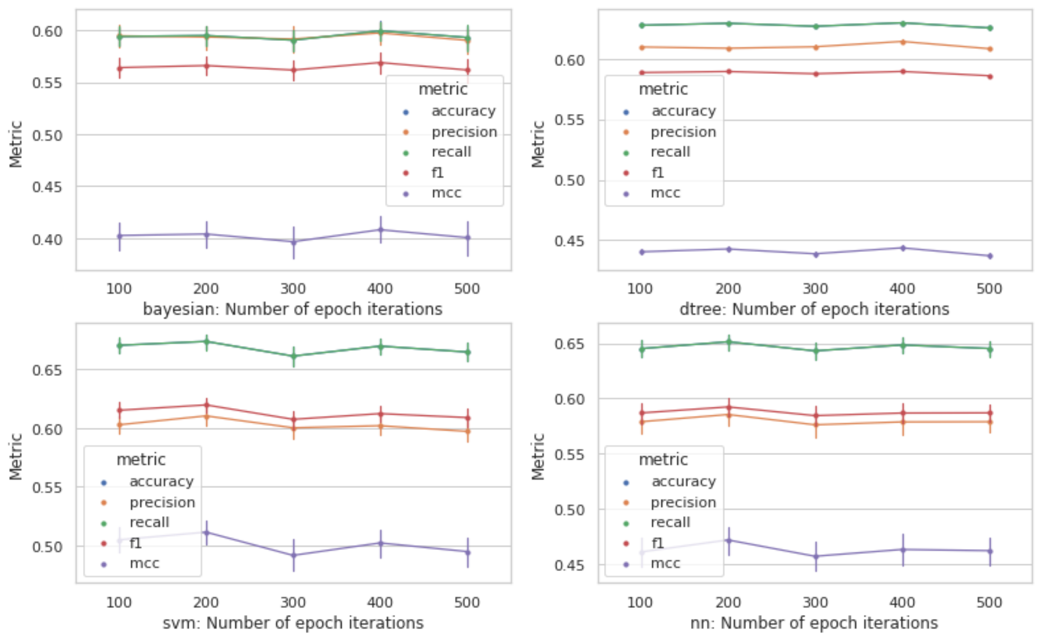

Figure 6.21: NSL-KDD Multiclass: Average performance across all combinations with an autoencoder against number of epochs trained

Figure 6.22 shows how varying the number of latent feature dimensions effects the overall performance average across all models. The maximum average performance is at 14 latent feature dimensions. As the number of latent dimensions increase, overall so does the performance of the ML classifiers.

Figure 6.23 displays the average effect on performance of including a PCA threshold before the data is passed into an autoencoder. This graph shows that performance definitely increases on average when some form of PCA thresholding is used. However, it is inconclusive if there is some optimal amount.

PCA Variance Threshold

Figure 6.24 displays the average effect on performance when including a PCA threshold before using ML classifiers. There doesn't seem to be any meaningful relationship between the average performance and PCA threshold.

6.3.2 Classification Evaluations

For each model type, this subsection will cover the effect dimensionality reduction has on the performance of them.

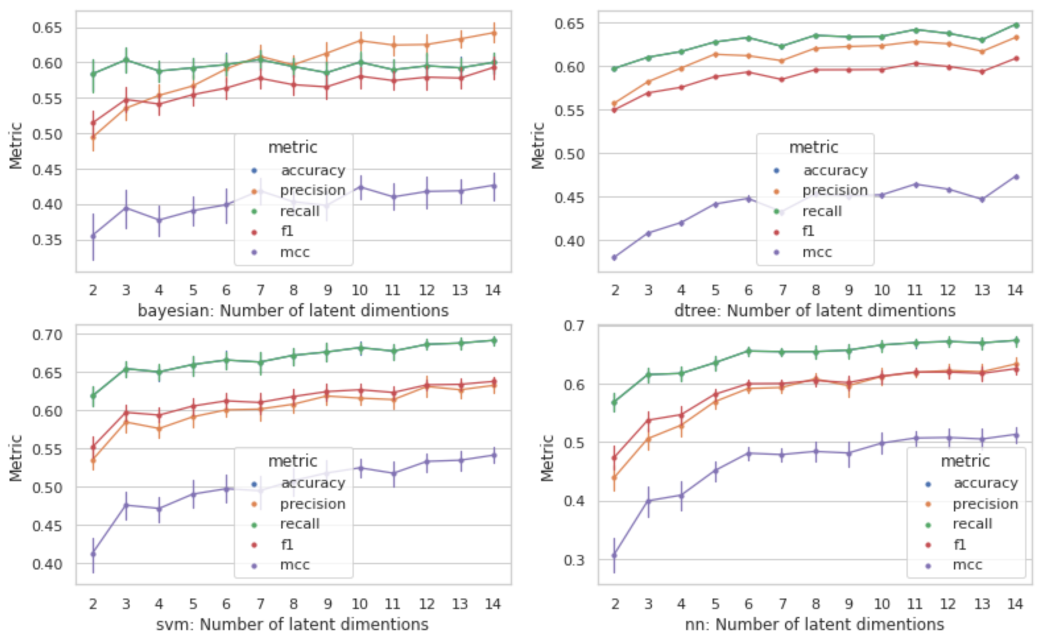

Figure 6.22: NSL-KDD multiclass: Average performance across all combinations with an autoencoder against number of latent feature dimensions

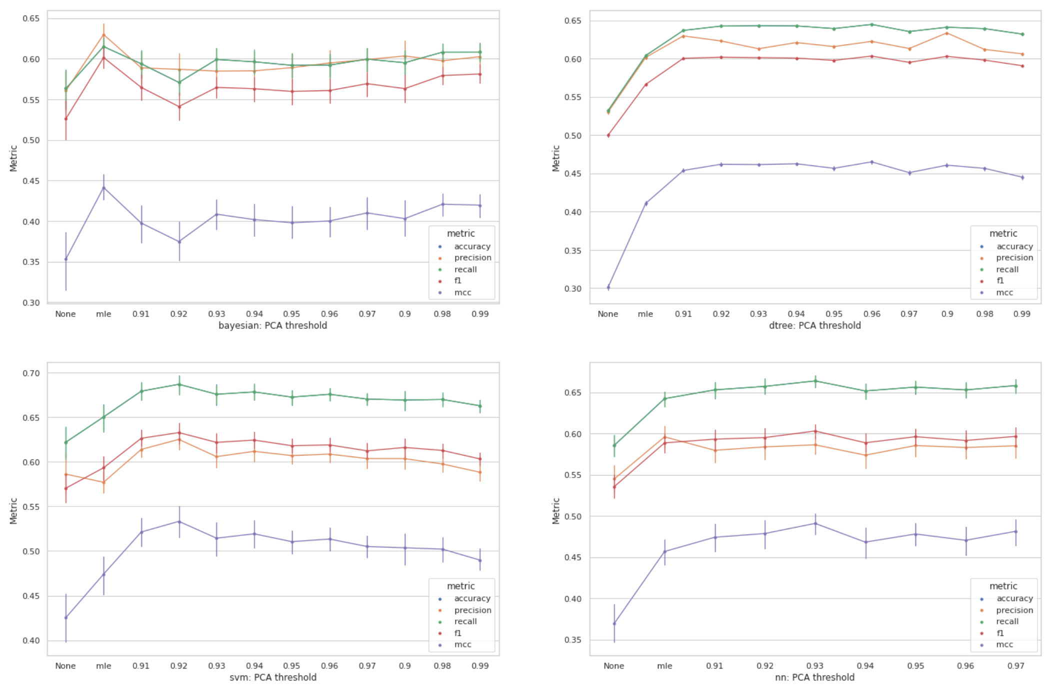

Figure 6.23: NSL-KDD Multiclass: Average performance across all combinations with an autoencoder against type of PCA variance threshold

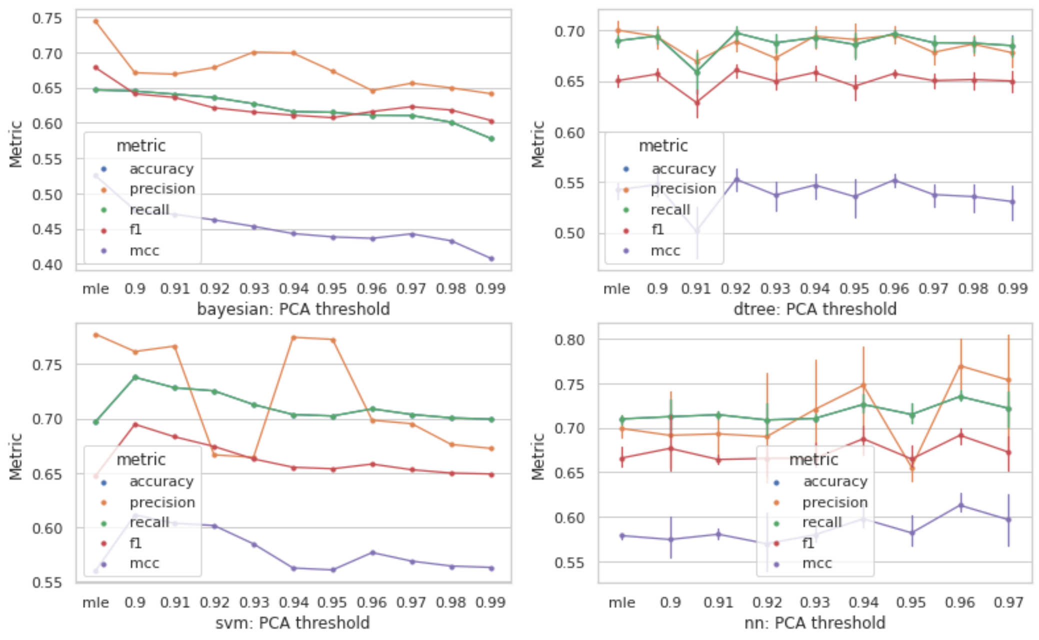

Figure 6.24: NSL-KDD Multiclass: Average performance across all combinations of ML classifier against type of PCA variance threshold

Table 6.13: NSL-KDD Multiclass: Mean improvement for each dimensional reduction technique from base metric for Gaussian Naïve Bayesian Networks

DR type	VAE	PCA	PCA + VAE
Accuracy	0.427858	0.484963	0.461766
Precision	0.290315	0.404595	0.326050
Recall	0.427858	0.484963	0.461766
F1	0.376720	0.474992	0.418575
MCC	0.451066	0.551061	0.504642

Gaussian Naïve Bayesian

Figure 6.25 shows the effect of the three different types of dimensional reduction on each of the performance metrics. The line in blue is the metric showing the performance of the model without any DR techniques applied.

This figure shows that each of the three DR methods produce some improvement in all of the metrics. Table 6.13 displays the mean increase in performance for each metric and DR technique.

The dimensional reduction technique that produced the highest overall metrics is a variational autoencoder with a PCA threshold trained for 300 epochs, with 10 latent feature dimensions, and with a threshold of 0.94. Table 6.14 shows the maximum

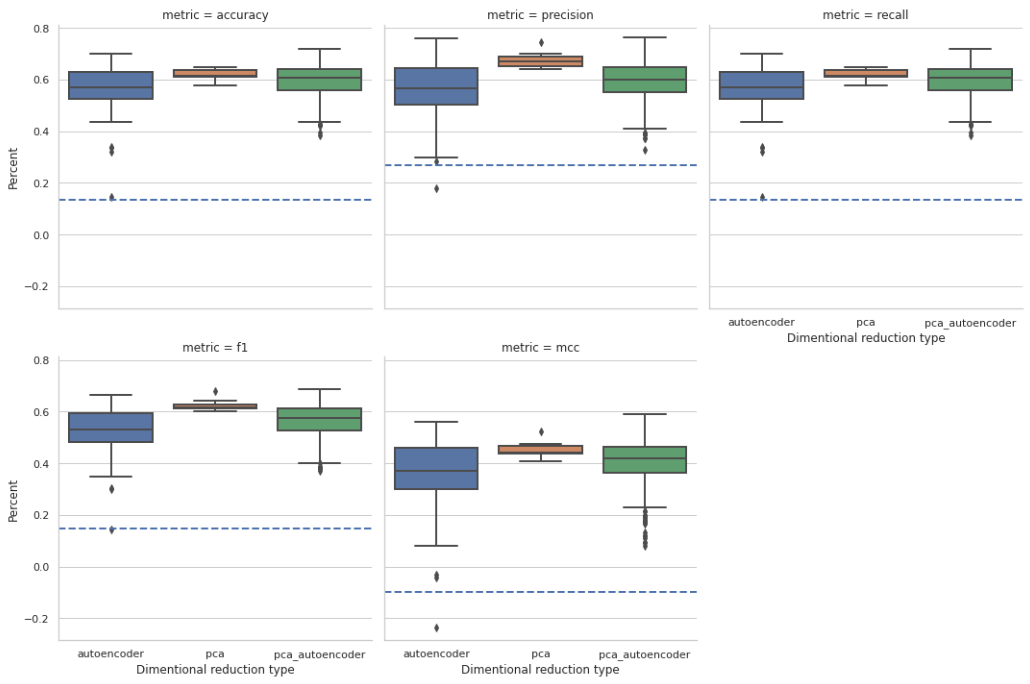

Figure 6.25: NSL-KDD Multiclass: Metric boxplot for all types of metric for Gaussian Naïve Bayes split by dimensional reduction method

Table 6.14: NSL-KDD Multiclass: Maximum improvement for each dimensional reduction technique from base metric for Gaussian Naïve Bayesian Networks

Meta DR type	DR	Accuracy	Precision	Recall	F1	MCC
VAE	400 epochs 13 features	0.699344	0.730050	0.699344	0.665330	0.560505
PCA	MLE threshold	0.646425	0.743603	0.646425	0.678121	0.525048
None	none	0.135380	0.270356	0.135380	0.149372	-0.097764
PCA w VAE	0.94 threshold 300 epochs 10 features	0.717308	0.725816	0.717308	0.681839	0.590013

performing dimensional reduction techniques for each type of DR.

Support Vector Machine with RBF Kernel

Figure 6.26 displays each of the various metrics for a SVM and the effect on them from altering the DR method. On average there was a decrease in performance from each of the DR methods, but there are instances from all three methods where the model gernated had better performance than the base model. Table 6.15 shows the average difference for each meta type of DR.6.16 show the maximum performing DR technique for each meta DR type.

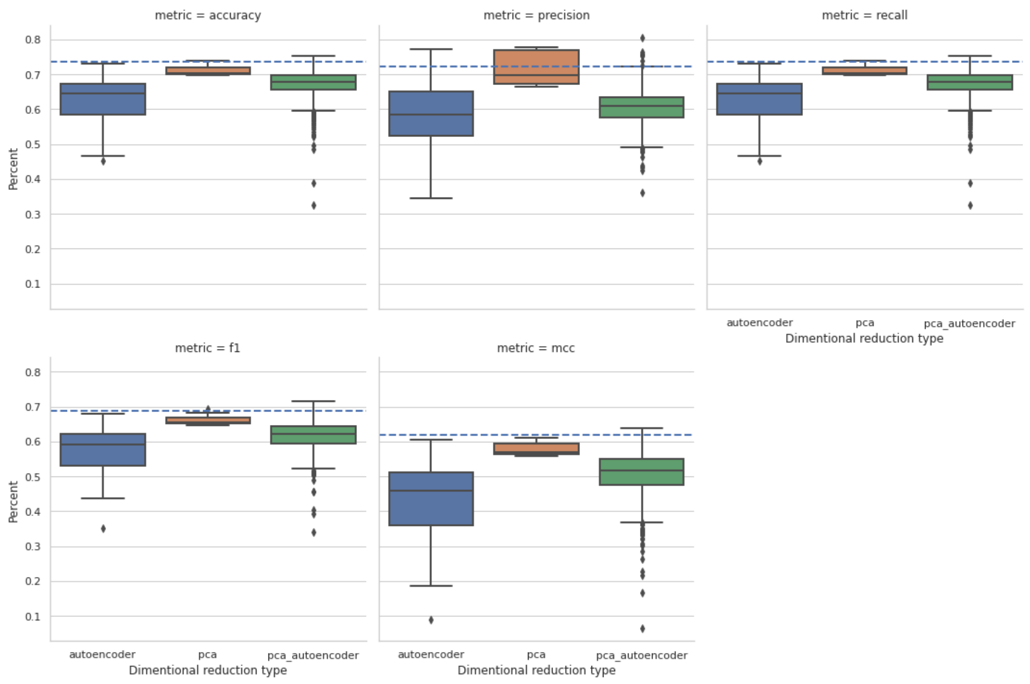

Figure 6.26: NSL-KDD Multiclass: Metric boxplot for all types of metric for SVM RBF kernel split by dimensional reduction method

Table 6.15: NSL-KDD Multiclass: Mean improvement for each dimensional reduction technique from base metric for SVMs with RBF kernel

DR type	VAE	PCA	PCA + VAE
Accuracy	-0.113277	-0.024502	-0.063438
Precision	-0.135341	-0.001002	-0.117678
Recall	-0.113277	-0.024502	-0.063438
F1	-0.116422	-0.025090	-0.070554
MCC	-0.193684	-0.041226	-0.111195

Table 6.16: NSL-KDD Multiclass: Maximum improvement for each dimensional reduction technique from base metric for SVMs with RBF kernel

Meta DR type	DR	Accuracy	Precision	Recall	F1	MCC
VAE	400 epochs 13 features	0.729950	0.659580	0.729950	0.678563	0.604440
None	none	0.735318	0.721507	0.735318	0.686905	0.619100
PCA	0.9 threshold	0.737802	0.761523	0.737802	0.694608	0.611262
PCA w VAE	0.92 threshold 500 epochs 7 features	0.750532	0.805002	0.750532	0.715518	0.638702

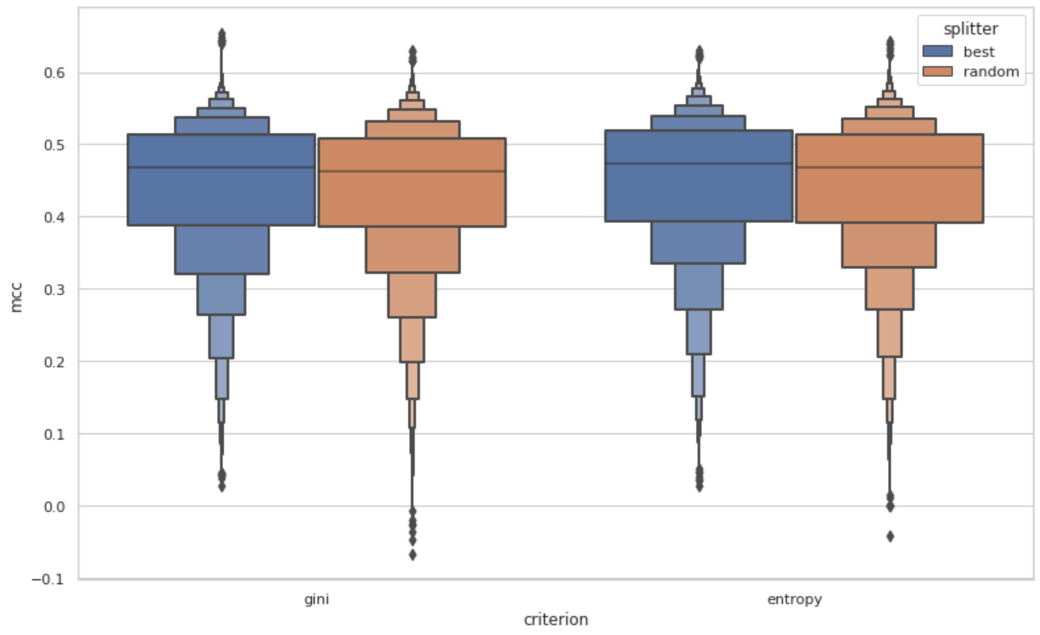

Figure 6.27: NSL-KDD Multiclass: Boxen plot of decision tree MCC split by classification criterion and node splitting strategy

Table 6.17: NSL-KDD Multiclass: Mean improvement for each dimensional reduction technique from base metric for Decision Trees

Criterion	Splitter	MSS	MSL	DR type	Accuracy	Precision	Recall	F1	MCC
gini	best	8	4	none	0.762598	0.816223	0.762598	0.723142	0.653463
	random	8	4	none	0.742770	0.806887	0.742770	0.697758	0.629296
entropy	best	2	3	none	0.746008	0.798570	0.746008	0.704365	0.629805
	random	2	2	none	0.750754	0.819936	0.750754	0.714789	0.642582

Decision Trees

As mentioned in section 4.3 there are 4 parameters that are varied with the decision tree model. 6.27 shows the effect on MCC of altering the `criterion` and `splitter` variables. There is very little different in the means for all four of the splits. Table 6.17 lists the maximum values for each split, but yet again there is very little difference. As a note, the dtree `gigi` and `best` parameters has the best performance out of any model for this dataset.

To compare the effectiveness of the DR techniques as accurately as possible, it is necessary that when observing their effectiveness we only change the DR technique, and not the model. With this set of decision trees there are 246 unique models each with different parameters. Instead of life in previous sections where we have created a

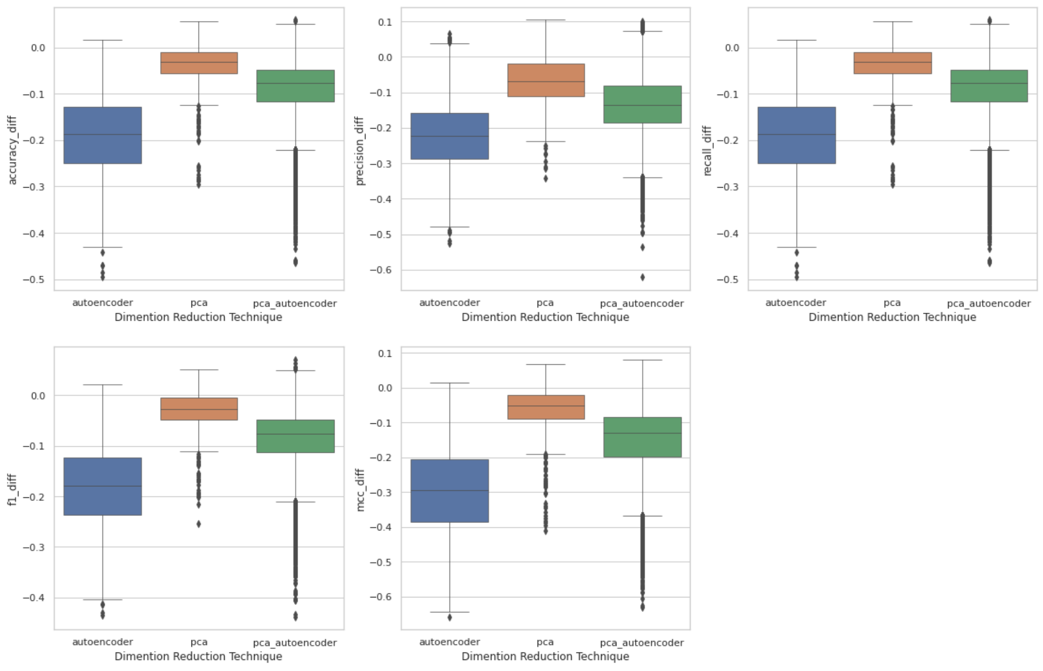

Figure 6.28: NSL-KDD Multiclass: Decision tree metric improvement box plots for each metric split by DR technique

set of box plots for a single model of each metric and specified the base metric, table 6.28 displays the distribution of the differences for each metric, combining each of the different model types into this visualisation.

The average improvement is negative for each DR technique, but each of the different classes have cases where the DR technique improves the performance of the model. The highest performing model with an accuracy of 0.762598 and a MCC of 0.653463 does not use any form of dimensional reduction, uses `gini` criterion and `best` splitting, and has a MSS of 8 and a MSL of 4.

Neural Networks

Figure 6.29 shows the performance of the dense neural network over the depth of the hidden layers without any DR techniques. On average it appears that using 1 or 3 layers gives the best performance as there is a massive dropoff at 5 layers.

Figure 6.30 shows distribution box plots for each of the metrics differences for each model. It shows that on average using any 3 of the techniques on average slightly increases the performance of the NNs, but there is large amount of variance (i.e. there are case where the performance of the models are both greatly increased and decreased). The neural network with the highest MCC and accuracy was a 1 layered model using

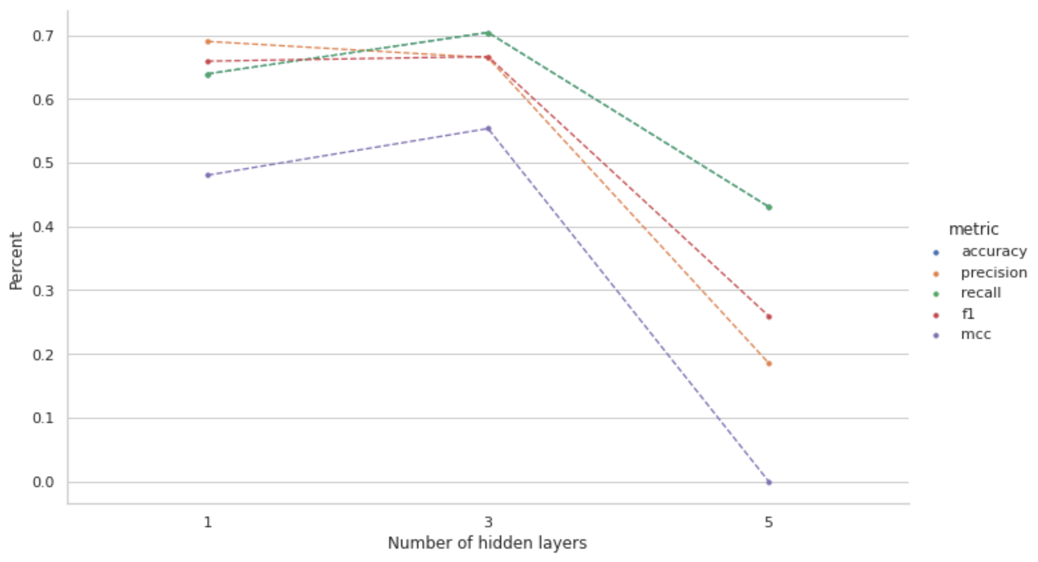

Figure 6.29: NSL-KDD Multiclass: Model performance over model hidden layer depth without DR

both PCA and a VAE with a threshold of 0.91 and trained with 100 epochs and 10 latent features.

Performance

Table 6.18 shows the top 10 performing models and the associated DR technique.

Table 6.18: NSL-KDD Multiclass: Top 10 performing models

DR type	Model type	Accuracy	Precision	Recall	F1	MCC
none	dtree gini best mss 2 msl 1	0.754214	0.772666	0.754214	0.705513	0.639363
PCA @ 0.93 autoencoder 2 6 VAE 300 epochs 6 features	dtree entropy random mss 8 msl 2	0.756920	0.686465	0.756920	0.705551	0.639674
none	dtree gini best mss 6 msl 4	0.751686	0.800946	0.751686	0.703585	0.640087
none	dtree gini best mss 8 msl 2	0.754480	0.793248	0.754480	0.705998	0.641928
none	dtree gini best mss 2 msl 3	0.753149	0.808719	0.753149	0.707224	0.642422
none	dtree entropy random mss 2 msl 2	0.750754	0.819936	0.750754	0.714789	0.642582
none	dtree gini best mss 4 msl 2	0.756166	0.806736	0.756166	0.709304	0.644231
none	dtree gini best mss 6 msl 3	0.756343	0.799394	0.756343	0.707705	0.644309
none	dtree gini best mss 4 msl 3	0.760468	0.780115	0.760468	0.721412	0.648953
none	dtree gini best mss 8 msl 4	0.762598	0.816223	0.762598	0.723142	0.653463

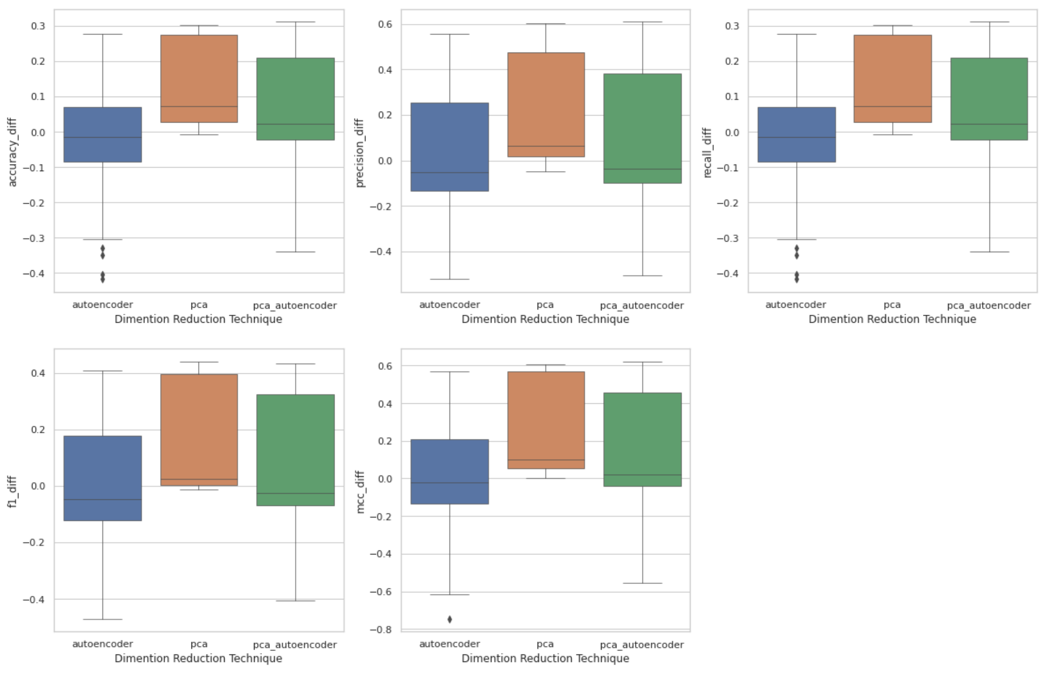

Figure 6.30: NSL-KDD Multiclass: Box plot of each metric improvement for neural networks split by DR technique

6.4 Discussion

For each of malware multiclass, intrusion binary, and intrusion multiclass datasets the following properties were analysed.

- How altering the parameters of the autoencoder effects the metrics of the classification models
- How altering the parameters of PCA thresholding effects the metrics of the classification models
- For each classification model, how altering the dimensional reduction method effects the metrics of the models.

From this, the effectiveness of variational autoencoders can be observed with respect to other dimensional reduction techniques across a multitude of machine learning algorithms.

To ensure the autoencoders were not under/over-trained the number of epochs trained will be varied and the resulting metrics were observed. The number of latent features have also be varied.

When constructing the results for the 3 different datasets, each of them produced vastly different results for each ML classifier. For the malware multiclass dataset the SVM with a RBF kernel with no dimensional reduction had the best performance (accuracy 0.978381 and MCC 0.973872). For the NSL-KDD binary dataset SMV with a RBF kernel again had the best performance, but it instead used both PCA and a VAE for DR (accuracy 0.828070 and MCC 0.682102). The highest performing model for the NSL-KDD multiclass dataset was a decision tree using gini entropy as a criterion, using the `best` splitting method, with a MSS of 8 and a MSL of 4 without an DR (accuracy 0.762598 and MCC 0.653463).

6.4.1 Malware Multiclass

Dimensional Reduction Parameter Tuning

In section 6.1.1 the results of altering the parameters of the dimensional reduction methods are displayed.

For VAEs it shows that both increasing the amount of training and increasing the number of latent dimension also increases the average performance of models trained on the latent features produced. It is interesting to note that figure 6.2 displays how restricting the number of latent reduces the accuracy, but for decision trees it only reduces the accuracy to 80% at with 2 latent features. This means that the VAE is able to compress 256 features [1] down to 2, and only lose around 15% accuracy performance. From figure 6.1 it seems that 500 epochs was almost enough to reach the maximum performance of the VAE features generation as at that point, the graph starts to level out for each metrics.

Figure 6.3 displays how adding a PCA threshold before a VAE affects average performance of each model. For this dataset, it seems that adding any threshold decreases performance.

Figure 6.4 displays how varying the strength of the PCA threshold affects performance. For this dataset it seems that having a higher overall threshold yields the best performance (However, as will be mentioned later, for this dataset PCA on average decreases performance when applied on this dataset).

[1] Although this is the total number of possible instruction sets, not all are used so the number of features actually used is less than this.

ML Classifiers performance with Dimensional Reduction Techniques

Figure 6.5 displays the effect on each performance metric produced from a trained Gaussian naïve Bayes network model when different DR techniques are used on the dataset. Out of the three methods just using a VAE produces the best improvement of performance, on average increasing accuracy by 8.07% and MCC by 8.96%. Just using PCA only ever decreased performance. Using a combination of the two on average increased performance although less than just a VAE but this combination also had a much higher variance, producing both the lowest performing model and the highest performing model (with a accuracy of 0.896964 and a MCC of 0.875812).

Using DR techniques with a SVM model with a RBF kernel was ineffective, not producing an improvement in performance out of any of the resulting models. SVM with no DR techniques did produce the highest performing model with an accuracy of 0.978381 and a MCC of 0.973872. However, four SVM RBF models including the top performing one were also in the top 10 performing models out of all the models trained. This is impressive as those 3 models were using a VAE producing 11, 13 and 14 features and were still able to produce high quality classification with much fewer dimensions to work on.

Figure 6.8 displays that on average none of the DR techniques increased the performance of the decision tree models. However there were some instances where the performance did increase, but none these surpassed the highest performing model not using DR with `entropy`, `best`, `mss = 6`, and `msl = 1` parameters set. This was believed to be the case due to the use of the `msl` and `mss` parameters which perform a similar function to DR techniques by attempting to ignoring outlier cases which causes the model to become more generalised.

The neural networks produced high performing models, with the 5 layer, 3 layer, and 1 layer taking the 2nd, 4th, and 3th places in the top 10 performing models respectively. As this is the case, none of the DR using models outperformed their plain counterpart. This is believed to be the case primarily due to a flaw in how the models were constructed. This will be discussed in section 6.5.1.

6.4.2 NSL-KDD Binary

Dimensional Reduction Parameter Tuning

In section 6.2.1 the results of altering the parameters of the dimensional reduction methods are displayed.

Initially number of epochs trained are plotted against average metric performance for each type of model. From the resulting figure 6.11 it appears that the VAEs were mostly fully trained by only 100 epochs. Training further however did produce a better average performance, so only the 500 epoch trained networks were used for further analysis. Figure 6.12 looks that the average performance of the metrics for each type of model when varying the number of latent dimensions the VAE used produces. On average the accuracy of the models increase as the number of latent dimensions increase, however this trend is quite loose with all models apart from neural networks the reason for this is discussed in section 6.5.1. Finally for VAEs, the effect of adding a PCA threshold before the VAE is examined with figure 6.13 showing the results. In every model, adding some PCA threshold before a VAE improved the average performance compared to a VAE with no threshold. It is inconclusive however if there is some optimal threshold amount.

Figure 6.14 is used to show the relationship between average performance for each model when varying the strengths of a PCA threshold. It appears there is no discernible trend.

ML Classifiers performance with Dimensional Reduction Techniques

Figure 6.15 displays the effect on each performance metric produced from a trained Gaussian naïve Bayes network model when different DR techniques are used on the dataset. All three methods produce improvements when used with a Bayesian network. Although recall drops, precision increases much more, and as a result the more comprehensive metrics such as accuracy, F1, and MCC increase. Using a VAE with PCA produces the best improvement on average out of the three and also produced the highest performing model (see tables 6.7 and 6.8).

When using any of the three DR methods with a SVM RBF model, on average none of them produce any improvements in performance (show in figure 6.16). However there were instances where the maximum performing models for each of the 3 DR methods outperformed the base model (show in table 6.10). The maximum performing model used a PCA with a VAE with 6 features and a 0.92 PCA threshold, and this a achieved an accuracy of 0.828070 and a MCC of 0.682102. This was the highest performing model overall.

Figure 6.8 displays that on average none of the DR techniques increased the performance of the decision tree models. However there were some instances where the performance did increase, but none these surpassed the highest performing model not using DR with `gini`, `random`, `mss = 6`, and `msl = 3` parameters set. This was believed to be the case due to the use of the `msl` and `mss` parameters which perform a similar function to DR techniques by attempting to ignoring outlier cases which causes the model to become more generalised.

Compared to the performance the neural networks had on the malware dataset, they performed extremely poorly on this dataset. Using DR techniques also decreased on average the performance of the neural networks, although there were cases where performance was increased (shown in table 6.19). The best performing NN model had only one layer and used a PCA threshold of 0.91 and a VAE trained to 500 epochs with 11 latent features.

6.4.3 NSL-KDD Multiclass

Dimensional Reduction Parameter Tuning

In section 6.3.1 the results of altering the parameters of the dimensional reduction methods are displayed.

Figure 6.21 shows the effect on average performance as the number of epochs the VAE is trained for is varied. From each of the graphs it appears that the autoencoder is already mostly trained at 100 epochs as there is very little difference between the metrics, and they even decreases at the higher epochs. For this section all the VAEs of different epochs will be analysed.

Figure 6.22 shows how performance is affected by varying the number of latent features the VAE produces. In general as the number of latent features increase so does the average performance of the models trained on those features.

Figure 6.24 is used to show the relationship between average performance for each model when varying the strengths of a PCA threshold. For all model types apart from neural networks, it appears that performance is increased as the threshold is tightened. This relationship is very weak however and there are various exceptions to this trend.

ML Classifiers performance with Dimensional Reduction Techniques

Figure 6.25 displays the effect on each performance metric produced from a trained Gaussian naïve Bayes network model when different DR techniques are used on the dataset. All three methods produce improvements when used with a Bayesian network. Just using a PCA threshold produces the best improvement on average out of the three, but using PCA with a VAE produced the highest performing Bayesian model (see tables 6.13 and 6.14).

When using any of the three DR methods with a SVM RBF model, on average none of them produce any improvements in performance (show in figure 6.26). However there were instances where the maximum performing models for each of the 3 DR methods outperformed the base model (show in table 6.16). The maximum performing model used a PCA with a VAE with 7 latent features and a 0.92 PCA threshold, and this a achieved an accuracy of 0.750532 and a MCC of 0.638702

Figure 6.28 displays that on average none of the DR techniques increased the performance of the decision tree models. However there were some instances where the performance did increase, but none these surpassed the highest performing model not using DR with `gini`, `best`, `mss = 8`, and `msl = 4` parameters set. This was believed to be the case due to the use of the `msl` and `mss` parameters which perform a similar function to DR techniques by attempting to ignoring outlier cases which causes the model to become more generalised.

Compared to the performance the neural networks had on the malware dataset, they performed extremely poorly on this dataset. Using DR techniques also decreased on average the performance of the neural networks, although there were cases where performance was increased (shown in table 6.29). The best performing NN model had only one layer and used a PCA threshold of 0.91 and a VAE trained to 500 epochs with 10 latent features.

6.5 Evaluation

The original aim of this study was defined as:

"This study attempts to determines the effectiveness of using variational autoencoders to produce latent feature representations of data in order to increase the accuracy of machine learning algorithms. Specifically, neural networks, Bayesian networks, support vector machines, and decision trees are trained and tested with malware and intrusion detection datasets."

This project was able to:

- Construct useable datasets from a variety of sources
- Train multiple VAEs via unsupervised learning while varying the number of latent features they produce

- Train each of the neural network, Bayesian networks, SVM, and decision tree models on the generated features created by the VAEs as well as the raw data
- Evaluate the performance of each of these models via the generation of multiple metrics
- Compare the metrics of the models with and without the various DR techniques applied

Through these steps the aim of the project was completed.

6.5.1 Strengths & Weaknesses

When constructing the dense neural networks the structure of the network was dependent on the size of the input and outputs. As a result of this, smaller networks were created when less features were used. Because of this, the results generated were not only dependent on the amount of "information" contained in the features the models were trained on, but also the number of input features and output classes. This caused results to be biased against DR methods that reduced the feature count. As an example, the effect of this can be seen in figure 6.4, where all other models apart from neural networks are improved by adding a PCA threshold. Another effect of these smaller networks is the increase chance to get stuck in a local minimum (see section 2.3.4). This happened very frequently in NSL-KDD binary due to the lower number of output features and as a result the models affected would always predict the test packets as "benign". To attempt to mitigate this problem an alternative optimizer was used to train the smaller networks which used a momentum feature. This reduced the number of faulty models, but there was still a significant number generated.

When training the various models each parameter was varied. This was to try and gather a better understanding on how effective the various DR methods were on the different models. For example, if all the parameters had been left on default it would be possible that a conclusion could be drawn that VAEs *always* increase the performance, which this project found to be incorrect. However, due to the large amount of data generated it was very hard to find visualisations that fully represented the effect on performance from each of these variables. This report gives a good generalisation of the overall effect of the DR methods, but it is lacking on the more in depth relationships and features of each model.

By including multiple metrics the performance of each model can be more precisely gauged. When selecting the best performing models, often there would be models that produced the same accuracy statistic. However, as MCC also accounts for the size of each class, it is more precise than accuracy and F1 and so a definitive best model was able to be chosen at every point.

The scope of this project was fairly large. As a result the specifics of performance of each model is not covered (i.e. How well each model classifies new types of data) but instead more of a big picture view is displayed. This is definitely beneficial at this stage of work, as not much work has been done into the potential of using VAEs in this cybersecurity context. This larger view provides a good baseline and guide for further work to be carried out.

6.6 Comparison

Table 6.19: Comparison of other papers models that were also trained on the Microsoft Malware dataset

DR type	ML Model	Evaluation Method	Accuracy	Citation
unigram representation	SVM RBF	5-cross validation	95.6	Yousefi-Azar et al. 2017
3 layered Autoencoder with 10 LFDs	SVM RBF	5-cross validation	96.3	Yousefi-Azar et al. 2017
unigram representation	K-NN	5-cross validation	94.0	Yousefi-Azar et al. 2017
3 layered Autoencoder with 10 LFDs	K-NN	5-cross validation	96.0	Yousefi-Azar et al. 2017
unigram representation	Naïve Bayes	5-cross validation	66.2	Yousefi-Azar et al. 2017
3 layered Autoencoder with 10 LFDs	Naïve Bayes	5-cross validation	80.4	Yousefi-Azar et al. 2017
unigram representation	Xgboost	5-cross validation	98.2	Yousefi-Azar et al. 2017
3 layered Autoencoder with 10 LFDs	Xgboost	5-cross validation	95.7	Yousefi-Azar et al. 2017
ASM instruction frequency	Random Forest	10-cross validation	95.7	Hassen, Carvalho, and Chan 2017
Opcode 2-gram	Random Forest	10-cross validation	99.21	Hassen, Carvalho, and Chan 2017
Opcode 2-gram using control stmt shingling	Random Forest	10-cross validation	99.21	Hassen, Carvalho, and Chan 2017
Opcode Frequency	Random Forest	3-cross validation	99.48	Rathore et al. 2018
Variance Threshold	Random Forest	3-cross validation	99.78	Rathore et al. 2018
1 layer Autoencoder with 32 LFDs	Random Forest	3-cross validation	99.41	Rathore et al. 2018
3 layer Autoencoder with 32 LFDs	Random Forest	3-cross validation	99.36	Rathore et al. 2018
None	SVM RBF	20% test train split	97.83	(This paper)
None	CART Decision Tree	20% test train split	96.82	(This paper)

Table 6.20: Comparison of other papers models that were also trained on the NSL-KDD dataset for binary classification evaluated on NSLTest+

DR type	ML Model	Accuracy	Citation
Autoencoder	SVM RBF	84.96	Al-Qatf et al. 2018
None	Recurrent NN	83.28	Yin et al. 2017
3 layered Autoencoder with 10 LFDs	Naïve Bayes	83.34	Yousefi-Azar et al. 2017
None	J48 dtrees	81.05	Tavallaee et al. 2009
None	Gaussian Naïve Bayes	76.56	Tavallaee et al. 2009
None	NBtree	82.02	Tavallaee et al. 2009
None	Random Forest	80.67	Tavallaee et al. 2009
None	Random Tree	81.59	Tavallaee et al. 2009
None	Multi-layer Perception	77.41	Tavallaee et al. 2009
None	Linear SVM	69.52	Tavallaee et al. 2009
PCA @ 0.92 & VAE 6 LFD	SVM RBF	82.81	(This paper)
PCA @ 0.97 & VAE 4 LFD	Gaussian Naïve Bayes	82.15	(This paper)

Table 6.21: Comparison of other papers models that were also trained on the NSL-KDD dataset for multiclass classification evaluated on NSLTest+

DR type	ML Model	Accuracy	Citation
Autoencoder	SVM RBF	80.48	Al-Qatf et al. 2018
None	Gaussian Naïve Bayes	74.40	Al-Qatf et al. 2018
None	J45 Decision Tree	74.60	Al-Qatf et al. 2018
None	NB-tree	75.40	Al-Qatf et al. 2018
None	Multi Layer Perceptron	78.10	Al-Qatf et al. 2018
None	Recurrent NN	81.29	Yin et al. 2017
None	CART Decision Tree	76.26	(This paper)
PCA @ 0.93 & VAE 6 LFD	CART Decision Tree	75.69	(This paper)

Chapter 7

Conclusion

For each dataset, decision trees, SVMs, Naïve Bayesian networks, and dense neural networks were trained with different dimensional reduction techniques. The DR techniques used were, PCA thresholding, Convolutional Variational Autoencoders, and a combination of the two. A base metric was also calculated by not using any of the DR methods. This process was repeated three times on different datasets.

It was found that when using VAEs to generate latent features with the Microsoft malware dataset: when using suitable VAEs with Gaussian naïve Bayesian networks performance was increased above the original values. The maximum performing model of this type outperformed the base model by 13% to reach an accuracy of 0.896964. All DR methods were unable to improve the performance of SVM RBF models. VAEs were able to improve the performance of some decision tree models, but were unable to improve the maximum performance.

With the NSL-KDD binary dataset: Again, VAEs consistently improved the performance of Gaussian naïve Bayesian networks producing the second highest performing model with this simple model type (Using a VAE with PCA with an accuracy of 0.821505). When using suitable VAEs with SVM RBF performance was increased above the original values. The maximum performing model of this type outperformed the base model by 8% to reach an accuracy of 0.828070 which was the maximum for this dataset. VAEs were able to improve the performance of some decision tree models, but were unable to improve the maximum performance.

With the NSL-KDD multiclass dataset: VAEs consistently improved the performance of Gaussian naïve Bayesian networks producing a maximum performance increase of 68% to accuracy. When using suitable VAEs with SVM RBF performance was increased above the original values. The maximum performing model of this type outperformed the base model by 2% to reach an accuracy of 0.750532. VAEs were able to improve the performance of some decision tree models, but were unable to improve the maximum performance.

As described in section 6.5.1, experiments with the dense neural networks were inconclusive.

7.1 Recommendations

- When testing decision trees there were no occasions when a model using a VAE outperformed the maximum performing base model. However, a large amount of parameter tuning was needed to find an ideal model. It is recommend that when using decision trees `mss` and `msl` are selected carefully to achieve the best possible performing model.
- Consistently throughout each experiment using a VAE with Gaussian Naïve Bayesian networks increased the performance consistently. It is very much recommended whenever this type of model is used, a VAE should be used to generate latent features for it.
- When constructing VAE it was often the case that using a PCA threshold increased the maximum performing model. It is recommended that a PCA threshold used before a VAE should be experimented with to increase model performance.

7.2 Future Work

One of the benefits of using a reduced number of features for training is the increased training time. However, this measure was not recorded in this paper. It would be worthwhile to formally determine how VAEs reduce this.

With the NSL-KDD dataset, there are sub categories contained in the test set that are not present in the training set. It would be interesting to if VAEs increase the likelihood of correctly classifying these new subcategories. This may be the case due to the nature of how VAEs are trained with the idea of similar samples should produce similar latent features (see section 2.3.4).

This report only looked at how VAEs compared to PCA and using no DR technique. It would be interesting to see how regular AEs compare to VAEs with the same datasets. This is partially covered in section 6.6 but does not go into depth.

Unfortunately, the experiments with neural networks were inconclusive but the experiments with them could be repeated but with the implementation altered to avoid altering the structure of them. This would hopefully yield worthwhile results.

Only the Gaussian Naïve version of Bayesian networks were experimented with in this report. Considering they were able to achieve the second highest performance with the NSL-KDD binary dataset, it seems it would be worthwhile to experiment with their capabilities further.

The depth of the VAEs was mentioned, but not explored. It would be worthwhile to look further into how varying the depth and structure of the VAE effects its performance.

VAEs were able to produce the highest performing models in the NSL-KDD binary dataset. It makes sense that from that there should be other datasets well suited for them. This experiment could be repeated for other datasets.

This report does not look into to accuracy of detecting malicious software. This is because of the extra amount of effort it would have been to collect and then decompile benign software to compare against the malware dataset. Future work could look at this problem.

7.3 Reflection

This projects direction was initially influenced by “Autoencoder-based feature learning for cyber security applications” (Yousefi-Azar et al. 2017). As a result, a similar number of ML classifiers were used. In reflection it would have been nice to have fewer of these classifiers so that more in depth analysis could have been done on each unique classifier. However, a benefit of having a larger number of these classifiers is the increase in knowledge gained from working with a broader spectrum of topics.

Initially when running the experiment it was programmed to run all in one go, creating all of the VAE networks at once then moving on to creating all the models at once, then finally generating the performance metrics. On the first run, it took a whole week to complete the multiclass malware part of the experiment and then hung, likely due to running out of memory. To attempt to mitigate this problem the program was rewritten, cutting the experiment into parts. First training and saving the models, and then a second program which trained the classification models and generated the metrics. This extra granularity meant that multiple machines were able run the experiment on multiple CPU cores and so it took around 6 days of computation time to run the remainder of the experiment. It would have been much smarter to have done this method at the start.

When training the VAEs, initially this was done on the MMU remote access machines. However, when the training had completed the `tensorflow` version that was used on the lab machines had a bug which made the saved models useless. Thankfully, this training was started very early on in the projects timeline, and so time was not an issue. To solve this issue the training was instead ran on personal machines detailed in table 5.1 using the newest version of `tensorflow`. This issue could have been avoided with better testing. After researching the bug later on, an alternative format could have been used for the saving of models.

It was only realised after the experimental results had been computed that the autoencoders that were trained on the NSL-KDD mluticlass and NSL-KDD binary datasets were the same, and could have been reused instead of training two separate times. This is because the VAE are trained via *unsupervised* learning, meaning that they do not get to observe any of the classes of samples. As this is the only difference between the two datasets, the VAEs used were the same.

The scope of this project was fairly large. As a result more in depth analysis of individual models was missed, but this allowed a greater quantity of techniques to be used. These techniques include using all the various ML classifiers that were used, supervised and unsupervised learning, a variety of data science python libraries. This as well as with the time spent learning about each technique when writing the literature review greatly expanded my knowledge.

References

Ahmad, Muhammad Aurangzeb, Carly Eckert, and Ankur Teredesai (2018). "Interpretable machine learning in healthcare". In: *Proceedings of the 2018 ACM international conference on bioinformatics, computational biology, and health informatics*, pp. 559–560.

Berrar, Daniel and Werner Dubitzky (2013). "Information Gain". In: *Encyclopedia of Systems Biology*. Ed. by Werner Dubitzky et al. New York, NY: Springer New York, pp. 1022–1023. ISBN: 978-1-4419-9863-7. DOI: 10.1007/978-1-4419-9863-7_719. URL: https://doi.org/10.1007/978-1-4419-9863-7_719.

Chacon, Scott and Ben Straub (2014). *Pro git*. Apress.

Cortes, Corinna and Vladimir Vapnik (1995). "Support-vector networks". In: *Machine learning* 20.3, pp. 273–297.

DeepMind (2016). *AlphaGo - The story so far*. URL: https://deepmind.com/research/case-studies/alphago-the-story-so-far (visited on 08/05/2021).

F.R.S., Karl Pearson (1901). "LIII. On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11, pp. 559–572. DOI: 10.1080/14786440109462720.

Foundation, Python Software (2016). *Python Language Reference, version 3.9.6*. URL: https://docs.python.org/3/reference/ (visited on 08/05/2021).

Hassen, Mehadi, Marco M Carvalho, and Philip K Chan (2017). "Malware classification using static analysis based features". In: *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, pp. 1–7.

Hunter, J. D. (2007). "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3, pp. 90–95. DOI: 10.1109/MCSE.2007.55.

Iformation and University of California Computer Science (1999). *KDD Cup 1999 Data*. URL: `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html` (visited on 08/12/2021).

Kelley, Henry J (1960). “Gradient theory of optimal flight paths”. In: *Ars Journal* 30.10, pp. 947–954.

Kluyver, Thomas et al. (2016). “Jupyter Notebooks – a publishing format for reproducible computational workflows”. In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides and B. Schmidt. IOS Press, pp. 87–90.

Liou, Cheng-Yuan et al. (2014). “Autoencoder for words”. In: *Neurocomputing* 139, pp. 84–96.

Liu, Jiwei (2015). *First place approach in Microsoft Malware Classification Challenge (BIG 2015)*. URL: `https://www.youtube.com/watch?v=VLQTRlLGz5Y` (visited on 09/05/2021).

Martıén Abadi et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: `https://www.tensorflow.org/`.

Microsoft (2009). *Malware Protection Center: Win32/Vundo*. URL: `https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan:Win32/Vundo` (visited on 08/24/2021).

— (2010). *Malware Protection Center: Win32/Simda*. URL: `https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan:Win32/Simda` (visited on 08/24/2021).

— (2011a). *Malware Protection Center: Win32/Ramnit*. URL: `https://web.archive.org/web/20130325055658/http://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=Win32/Ramnit` (visited on 08/24/2021).

— (2011b). *Malware Protection Center: Win32/Tracur*. URL: `https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan:Win32/Tracur` (visited on 08/24/2021).

— (2013a). *Adware:Win32/Lollipop*. URL: `https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Adware:Win32/Lollipop` (visited on 08/24/2021).

— (2013b). *Malware Protection Center: Win32/Gatak*. URL: `https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?`

Name=Trojan:Win32/Gatak%5C&threatId=-2147289564 (visited on 08/24/2021).

Microsoft (2014). *Malware Protection Center: Win32/Obfuscator.ACY*. URL: https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=VirTool:Win32/Obfuscator.ACY (visited on 08/24/2021).

— (2018). *Microsoft Malware Classification Challenge*. URL: http://arxiv.org/abs/1802.10135 (visited on 08/11/2021).

Minka, Thomas (2000). "Automatic choice of dimensionality for PCA". In: *Advances in neural information processing systems* 13, pp. 598–604.

Mitchell, Tom M et al. (1997). "Machine learning". In.

Murphy, Kevin P (2012). *Machine learning: a probabilistic perspective*. MIT press.

New Brunswick, University of (2021). *NSL-KDD Dataset*. URL: https://www.unb.ca/cic/datasets/nsl.html (visited on 08/15/2021).

NumFOCUS (2021). *pandas*. URL: https://pandas.pydata.org/ (visited on 08/05/2021).

NumPy (2021). *numpy*. URL: https://numpy.org/ (visited on 08/05/2021).

Ortloff, Stefan (2012). *FAQ: Disabling the new Hlux/Kelihos Botnet*. URL: https://securelist.com/faq-disabling-the-new-hluxkelihos-botnet/32634/ (visited on 08/24/2021).

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Al-Qatf, Majjed et al. (2018). "Deep learning approach combining sparse autoencoder with SVM for network intrusion detection". In: *IEEE Access* 6, pp. 52843–52856.

Rathore, Hemant et al. (2018). "Malware detection using machine learning and deep learning". In: *International Conference on Big Data Analytics*. Springer, pp. 402–411.

Saporito, Gerry (2019). *A Deeper Dive into the KSL-KDD Data Set*. URL: https://towardsdatascience.com/a-deeper-dive-into-the-nsl-kdd-data-set-15c753364657 (visited on 08/24/2021).

scikit-learn (2021a). *scikit learn decision tree documentation*. URL: https://scikit-learn.org/stable/modules/tree.html (visited on 09/13/2021).

— (2021b). *scikit learn metric documentation*. URL: https://scikit-learn.org/stable/modules/model_evaluation.html#matthews-corrcoef (visited on 09/01/2021).

Stuart, Alan (1994). “Kendall’s advanced theory of statistics”. In: *Distribution theory* 1.

Tavallaee, Mahbod et al. (2009). “A detailed analysis of the KDD CUP 99 data set”. In: *2009 IEEE symposium on computational intelligence for security and defense applications*. IEEE, pp. 1–6.

TensorFlow (2021). *CVAE tutorial*. URL: `https://www.tensorflow.org/tutorials/generative/cvae` (visited on 08/16/2021).

Vert, Jean-Philippe, Koji Tsuda, and Bernhard Schölkopf (2004). “A primer on kernel methods”. In: *Kernel methods in computational biology* 47, pp. 35–70.

Waskom, Michael L. (2021). “seaborn: statistical data visualization”. In: *Journal of Open Source Software* 6.60, p. 3021. DOI: `10.21105/joss.03021`. URL: `https://doi.org/10.21105/joss.03021`.

Yang, Xitong (2017). “Understanding the variational lower bound”. In: *variational lower bound, ELBO, hard attention*, pp. 1–4.

Yin, Chuanlong et al. (2017). “A deep learning approach for intrusion detection using recurrent neural networks”. In: *Ieee Access* 5, pp. 21954–21961.

Yousefi-Azar, Mahmood et al. (2017). “Autoencoder-based feature learning for cyber security applications”. In: pp. 3854–3861.

Appendix A

Example Code

```python
def log_normal_pdf(sample, mean, logvar, raxis=1):
  log2pi = tf.math.log(2. * np.pi)
  return tf.reduce_sum(
      -.5 * ((sample - mean) ** 2. * tf.exp(-logvar) + logvar + log2pi),
      axis=raxis)

def reparameterize(mean, logvar):
  eps = tf.random.normal(shape=mean.shape)
  return eps * tf.exp(logvar * .5) + mean

def sample(vae_model, eps=None):
  if eps is None:
    eps = tf.random.normal(shape=(100, vae_model.latent_dim))
  return vae_model.decode(eps, apply_sigmoid=True)

def encode(vae_model, x):
  pred = vae_model.encoder(x)
  mean, logvar = tf.split(pred, num_or_size_splits=2, axis=1)
  return mean, logvar

def decode(vae_model, z, apply_sigmoid=False):
  logits = vae_model.decoder(z)
  if apply_sigmoid:
    probs = tf.sigmoid(logits)
    return probs
  return logits

def compute_loss(vae_model, x):
```

```python
mean, logvar = vae_model.encode(x)
z = reparameterize(mean, logvar)
x_logit = vae_model.decode(z)
cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=x)
logpx_z = -tf.reduce_sum(cross_ent)
logpz = log_normal_pdf(z, 0., 0.)
logqz_x = log_normal_pdf(z, mean, logvar)
return -tf.reduce_mean(logpx_z + logpz - logqz_x)
```

Listing A.1: Loss function and reparameterization

Appendix B

ToR

Department of Computing and Mathematics Computing and Digital Technology Postgraduate Programmes Terms of Reference Coversheet	
Student name:	Thomas Taylor
University I.D.:	20105906
Academic supervisor:	Dr. Amna Eleyan
External collaborator (optional):	
Project title:	Using variational autoencoders with machine learning algorithms in cyber security applications
Degree title:	MSc Data Science
Project unit code:	6G7Z1029
Credit rating:	60
Start date:	01/06/2021
ToR date:	11/06/2021
Intended submission date:	24/09/2021
Signature and date student:	TTaylor 09/06/2021
Signature and date external collaborator (if involved):	

This sheet should be attached to the front of the completed ToR and uploaded with it to Moodle.

1 Aims

Determine the effectiveness of using variational autoencoders to produce latent feature representations of data in order to increase the accuracy of machine learning algorithms. Specifically, neural networks, Bayesian networks, support vector machines, and other machine learning methods are trained and tested with malware and intrusion detection datasets.

Research question

How does the addition of variational autoencoder for feature modification effect the accuracy of machine learning algorithms?

Impact

Determining the effect AVs have on ML algorithms will provide a way to possibly increase their accuracy/performance. This will allow for more accurate prediction/classification software applications.

Contribution

This project tests additional machine learning algorithms with variational encoders.

2 Objectives

- Terms of Reference - Providing a more detailed scope and description of the project
- Data Collation - Collation and preparation of the malware and intrusion detection datasets including basic analysis and preprocessing
- Data Analysis & Software Construction - Training and testing of the different machine learning methods including parameter tuning
- Final Report - Report encompassing all work done and information gathered

3 Leaning Outcomes

- Demonstrate a critical understanding of the challenges and issues arising from taking complex data, and applying data science techniques to gain insight for business, scientific or social innovation
- Develop the skills and expertise required for the analysis, interpretation and visualisation of complex, high-volume, high dimensional and structured/unstructured data from varying sources.
- Identify, apply, experiment with and evaluate appropriate machine learning algorithms to mine data and evaluate using statistical methods
- Have a systematic understanding of knowledge and a critical awareness of current problems relating to the field of network security and its deployment
- Undertake fundamental research and development activity related to network and security issues

4 Project Description

In the paper 'Autoencoder-based feature learning for cyber security applications' (Yousefi-Azar et al. 2017) a variable autoencoder was used in conjunction with various types of machine learning classifiers to separate malicious content from harmless content. In this paper an autoencoder was trained on two datasets (A malware dataset, and an intrusion detection dataset), then the various machine learning methods where then trained on the raw data and the latent feature representation that the autoencoder produces. The results were then compared.

This project continues a similar line of work. Variational autoencoders are trained on malware and intrusion datasets and the latent features that are produced are then used with more machine learning techniques. As an extension to previous work, additional machine learning algorithms are trained and tested. The results of this training and testing are analysed to determine if this initial feature modification from the variational autoencoder increases the accuracy of these additional machine leaning algorithms. It is assured that the use of each of the datasets will be lawful under their respective licenses.

5 Related works

This project was initially inspired from reading the following surveys and reviews.

- A survey of deep learning methods for cyber security (Berman et al. 2019)
- Machine learning and deep learning methods for cybersecurity (Xin et al. 2018)
- Machine learning and deep learning techniques for cybersecurity: a review (Salloum et al. 2020)

In addition to these papers, additional papers that share a similar cybersecurity/machine learning focus are collated and summarised and the results of those papers are compared against the results recorded in this project.

For example, the paper 'A Deep Learning Method With Filter Based Feature Engineering for Wireless Intrusion Detection System' (Kasongo & Sun 2019) uses a modified feed forward deep neural networks algorithm to perform classification on the same intrusion detection dataset. This algorithm is compared against various other machine learning algorithms and is found to outperform them with a binary classification accuracy of 87.74%.

6 Literature Review

A thorough literature is conducted consisting of perquisite topics on cybersecurity and machine learning:

- Intrusion Detection Systems (IDS)
- Malware methods and detection
- Neural Networks
- Variational Autoencoders
- Bayes' Theorem and Bayesian networks
- Support Vector Machines (SVM)
- etc...

7 Evaluation Plan

Each of the objectives are complete with the following items:

- Terms of Reference - Completed document detailing the scope and description of the project
- Data Collation - Prepared datasets with analysis and description of the dataset features
- Data Analysis & Software Construction - Results from training and testing from all of the machine learning algorithms and datasets
- Final Report - Completed report detailing the previous objectives and results obtained from these steps

To evaluate the effectiveness of using the variational autoencoder feature modification, the accuracy of each of the machine learning algorithms is compared both using the latent features and without. These results are also compared against similar experiments identified in the literature review.be

8 Activity Schedule

Task	Start Date	End Date	Duration (days)
Tor & Ethics	01/06/2021	11/06/2021	10
Literature Review	01/06/2021	31/07/2021	60
Data Collection	01/06/2021	09/07/2021	38
Data Analysis	09/07/2021	13/08/2021	35
Report Writing	13/08/2021	24/09/2021	42

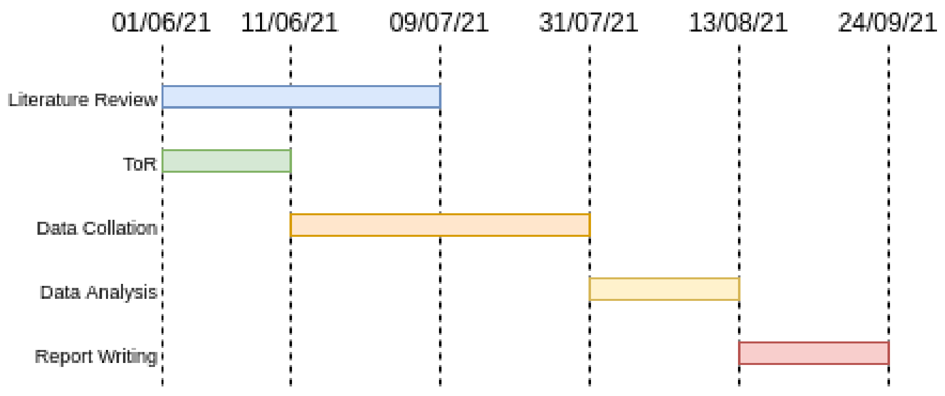

Figure 1: Gantt chart of project timeline

References

Berman, D. S., Buczak, A. L., Chavis, J. S. & Corbett, C. L. (2019), 'A survey of deep learning methods for cyber security', *Information* **10**(4), 122.

Kasongo, S. M. & Sun, Y. (2019), 'A deep learning method with filter based feature engineering for wireless intrusion detection system', *IEEE Access* **7**, 38597–38607.

Salloum, S. A., Alshurideh, M., Elnagar, A. & Shaalan, K. (2020), Machine learning and deep learning techniques for cybersecurity: a review, *in* 'Joint European-US Workshop on Applications of Invariance in Computer Vision', Springer, pp. 50–57.

Xin, Y., Kong, L., Liu, Z., Chen, Y., Li, Y., Zhu, H., Gao, M., Hou, H. & Wang, C. (2018), 'Machine learning and deep learning methods for cybersecurity', *Ieee access* **6**, 35365–35381.

Yousefi-Azar, M., Varadharajan, V., Hamey, L. & Tupakula, U. (2017), Autoencoder-based feature learning for cyber security applications, *in* '2017 International joint conference on neural networks (IJCNN)', IEEE, pp. 3854–3861.