





Trabajo Fin de Máster

Characterisation and Forecasting of Honeypot Data based on Machine Learning Techniques

Raquel Redondo Oubiña

Máster en Técnicas Estadísticas

Curso 2020-2021

Propuesta de Trabajo Fin de Máster

Título en galego: Caracterización e predición de datos de Honeypot baseados en técnicas de aprendizaxe automática
Título en español: Caracterización y predicción de datos de Honeypot basados en técnicas de aprendizaje automático
English title: Characterisation and Forecasting of Honeypot Data based on Machine Learning Techniques
Modalidad: Modalidad B
Autor/a: Raquel Redondo Oubiña, Universidad de Santiago de Compostela
Director: Javier Roca Pardiñas, Universidad de Vigo
Tutora: Marta Sestelo Pérez, Gradiant
Breve resumen del trabajo: En este trabajo se exploraron, bajo la supervisión de la empresa Gradiant, varios algoritmos para la caracterización y predicción de datos obtenidos mediante Honeypots. Para ello se utilizaton técnicas de de aprendizaje automático supervisado y no supervisado, con el objetivo de extraer información sobre los datos que sirva para entender mejor los ataques llevados a Honeypots.

Don Javier Roca Pardiñas, Profesor de la Universidad de Vigo y doña Marta Sestelo Pérez, Responsable Técnico de Data Analytics & AI de Gradiant, informan que el Trabajo Fin de Máster titulado

Characterisation and Forecasting of Honeypot Data based on Machine Learning Techniques

fue realizado bajo su dirección por doña Raquel Redondo Oubiña para el Máster en Técnicas Estadísticas. Estimando que el trabajo está terminado, dan su conformidad para su presentación y defensa ante un tribunal.

En Santiago de Compostela, a 6 de septiembre de 2021.

El director:

Don Javier Roca Pardiñas

La tutora:

Doña Marta Sestelo Pérez

La autora:

Doña Raquel Redondo Oubiña

Acknowledgements

I would like to thank my tutor at Gradiant, Marta, without whom I would not be presenting this thesis. Thank you for your guidance and patience, especially in the moments when I was struggling the most.

I would also like to thank Dr. Javier Roca, for sharing his invaluable knowledge with me.

Many thanks to the cybersecurity team at Gradiant, for being so helpful and extra nice to me.

I would like to dedicate this thesis to Hurí and my family.

Contents

Abstract **xi**

Preface **xiii**

0.1 Gradiant xiii

0.2 Overview of the problem xiv

0.2.1 Description of Honeypots xv

0.2.2 Brief Machine Learning Overview xvi

1 Cluster Analysis **1**

1.1 Overview of clustering algorithms 2

1.1.1 K-Means Clustering 2

1.1.2 K-prototypes Clustering 5

1.1.3 DBSCAN Clustering 7

1.2 Application to honeypot data 13

1.2.1 Description of the data 13

1.2.2 K-means clustering results 13

1.2.3 K-prototypes clustering results 22

1.2.4 UMAP dimensionality reduction with DBSCAN clustering results 29

1.2.5 Remarks about the results in this chapter 42

2 Forecast prediction of number of attacks **45**

2.1 Overview of forecasting algorithms 45

2.1.1 Forecasting with Random Forests 45

2.1.2 Forecasting with GAMs 49

2.1.3 Evaluating forecast accuracy 54

2.2 Application to real data 55

2.2.1 Forecasting with Random Forests 57

2.2.2 Forecasting with GAMs 60

2.2.3 Remarks about the results in this chapter 64

Abstract

Resumen en español

En este trabajo se llevaron varios análisis utilizando técnicas de machine learning para analizar datos recolectados por un Honeypot. Específicamente se exploró la utilización de tres algoritmos de clustering, en el contexto de aprendizaje no supervisado, para la caracterización de los ataques realizados a un Honeypot. Los clústeres formados fueron analizados para tratar de encontrar las características más relevantes de los ataques, con el objetivo de ayudar a mejorar medidas de ciberseguridad. Tambien, se probaron dos algoritmos muy conocidos en regresión, en el contexto de aprendizaje supervisado, para realizar predicciones del número de ataques diarios que recibe un Honeypot. Los resultados se analizaron y se hizo una comparación para decidir el algoritmo con mayor capacidad de predicción.

English abstract

In this work, several analyses were carried out using machine learning techniques to analyse the data collected by a Honeypot. Specifically, the use of three clustering algorithms was explored, in the context of unsupervised learning, for the characterisation of the attacks carried out on a Honeypot. The clusters that were formed were analysed to find the most relevant characteristics of the attacks, with the aim of improving cybersecurity measures. Also, two well-known regression algorithms were tested, in the context of supervised learning, to make predictions of the number of daily attacks a Honeypot receives. The results were analysed, and a comparison was made to decide the algorithm with the highest predictive power.

Preface

0.1 Gradiant

Gradiant (Galician Research and Development Center in Advanced Telecommunications) is a private non-profit organisation focused on the generation of technologies in the fields of ICT (Information and Communications Technology) and R&D (Research and Development). Officially founded in 2007, Gradiant is backed by a board integrated by representatives of three Galician universities and eight of the most important companies in the region. With more than ten years of experience, the company comprises more than 100 highly qualified professionals, with more than 100 clients distributed across more than 20 countries.

Current Gradiant projects are being developed under four main areas: *Advanced Communications*, *Multimodal Information*, *Intelligent Systems* and *Security & Privacy* (see Figure 1). Each area focuses on developing new technologies and products to provide solutions to specific problems in various industries.

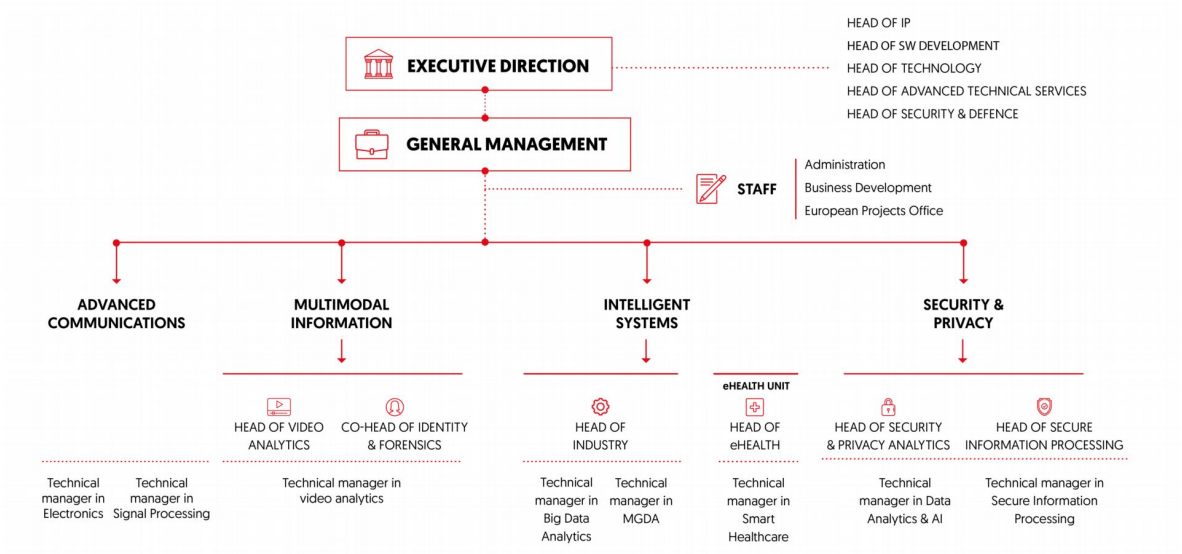


Figure 1: Organisational chart of Gradiant. Source: Gradiant web page.

The analyses and results presented in this work belong to the Security & Privacy area, which is

concerned with the development of cybersecurity technologies for networks, systems and information. The members of the cybersecurity team are responsible for the development of technologies that improve the security of systems through the use of secure communication systems, cryptographic techniques, and data analytics for the detection of anomalous events and the prevention of attacks carried out to systems. The Security & Privacy team is involved in the following technologies

- Advanced cryptographic systems for secure information processing.
- Biometric security.
- Data analytics for detection of anomalies.
- Protection of personal information.
- Secure communication systems.
- Security of infrastructures and borders through multimedia signal processing.

The analyses presented in this work were done with the aim to contribute to a tool that is being developed by the Security & Privacy team. This tool is in charge of the characterisation and prediction by analysing data of attacks made to a Honeypot. Hopefully, the results presented will help the tool by making better analyses of the data and improve the predictions of the number of attacks.

0.2 Overview of the problem

The use of machine learning techniques in the context of cybersecurity is becoming more relevant as the number of threats and vulnerabilities of systems rises as they become more complex. The constant increasing volume of cyberattacks urges automated mechanisms that are capable of identifying vulnerabilities, threats, and malicious activities in a timely manner. Machine learning algorithms are being employed to fight against cyber threats and attacks, with the aim of preventing as many future cyberattacks as possible or at least minimising their impact. There are many challenges, like anomaly detection and malware classification, where there is an increasing need for techniques that allow for data to be analysed. With the use of machine learning, cybersecurity systems can be analysed to find patterns and learn from them to help prevent threats and generate better countermeasure strategies.

The cybersecurity team at Gradiant is interested in exploring machine learning algorithms to analyse data collected by a honeypot (described in Section 0.2.1) in order to help security researchers and engineers to have a better understanding of the attacks. The principal objective is to analyse the data through the use of several machine learning algorithms. They are particularly interested in two tasks: the characterisation of the attacks and forecasting the number of attacks per day to a honeypot. For the first task, three different clustering algorithms were used to analyse the data. The second task was approached by using two regression algorithms in the forecasting context. With the results obtained through the use of these algorithms, the team at Gradiant will hopefully be able to gain some insight into the nature of attacks, draw better conclusions and gain knowledge in order to create improved strategies against cyberattacks and build more secure systems.

0.2.1 Description of Honeypots

A *honeypot* is a system to be placed on a network to act as bait with the purpose of having attackers probe, attack, and compromise it. The honeypot mimics a computer system to attract cyberattacks. They work by deceiving attackers into believing that they have gained access to a real system. Every step the attacker performs is logged in detail by the system, saving pertinent information about the attack like its total duration and the IP address of the attacker. The gathered data can be used by researchers and security professionals to monitor malicious attackers without their knowledge and learn about their techniques and exploits in order to prevent or counteract future attacks.

There are several ways to classify honeypots; however, the most common type of classification is based on the level of interaction between the intruder and the system. Based on this, honeypots can be classified into the following categories:

- *Low interaction honeypot*: Honeypots with limited interaction with the attacker. These honeypots do not have an operating system for attackers to interact with, but they attract attackers by using software to emulate features of a particular operating system and network services. This honeypot is easy to deploy and maintain, and does not involve complex architecture. Because of its simplicity, it may not respond accurately to exploits. One of its main objectives is to detect the source of unauthorised activity.
- *Medium interaction honeypot*: These type of honeypots are slightly more sophisticated than low interaction honeypots. Medium interaction honeypots emulate collections of software to give the attacker the illusion of targeting a real operating system. This allows for the system to handle more complex attacks, therefore capturing more detailed information about the attacks.
- *High interaction honeypot*: The most sophisticated level of honeypot given its interaction. It gives the attacker a more realistic experience by providing a higher level of interaction with the system; there is little to no restriction to what the attacker can do once they have comprised the system. These log and gather more information about the attacks. They are also more complex and challenging to design and maintain, and need more infrastructure and counter measures to prevent an attacker from misusing it.

There are commercial and open-source honeypots available for security professionals. All the datasets used throughout this work were obtained from the open-source *Cowrie* honeypot. *Cowrie* is a medium to high interaction honeypot designed to log brute force attacks and the interaction performed by the attacker. When it is configured as a medium interaction mode, it emulates a UNIX system in Python, see Oosterhof (2018) for more information. *Cowrie* provides a fake file system and a fake ssh shell so the attacker can interact with the system. It can be configured to emulate different types of operating system platforms. All gathered data is logged in JSON format, which can be processed to prepare the data for future analyses.

Depending on its configuration, a *Cowrie* honeypot has the following features:

- A fake filesystem with the ability to add and remove files.
- The possibility of adding fake contents to files, so the attacker can read them.
- Downloaded files can be saved with commands such as `wget`.

Overall, honeypots are systems used by cybersecurity experts to help them understand attacker behaviour patterns. This allows them to study system vulnerabilities which helps them improve the security of a system. In this work, we use the data gathered by honeypots to gain more insights about cyberattacks, and thus using this information to create more secure systems.

0.2.2 Brief Machine Learning Overview

Machine learning can be considered to be the process of applying a computer-based resource to implement learning algorithms for automatic pattern recognition and intelligent decision making based on training sample data. Informally, it can be regarded as the ability of machines to learn without being explicitly programmed. Machine learning algorithms can be categorised according to the task they perform and their learning type. There are four main classes of learning type:

- Supervised learning
- Unsupervised learning
- Semi-supervised learning
- Reinforcement learning

In supervised learning, an algorithm is fed sample data that are labelled in meaningful ways. The algorithm uses the labelled samples to train a model. Then, the trained machine learning model can be used to make predictions for new datapoints. The objective of using a supervised machine learning algorithm is to obtain the highest classification accuracy. In unsupervised learning, no target variable or labels are given in the sample data. Unsupervised learning algorithms work by summarising key features in the data and finding natural patterns in the data given a particular cost function. Semi-supervised learning falls between supervised and unsupervised learning. Semi-supervised algorithms use a small portion of labelled samples and a large number of unlabelled samples for training a model. It has high practical value especially in cases where labelled data is difficult to obtain or is very expensive. The last category, reinforcement learning, is the branch of machine learning which takes into consideration the finite states in an environment to learn and choose actions that will maximise some reward.

Because machine learning algorithms are used to build models that can analyse threats and respond to security incidents and attacks in an automated way, they are employed to tackle many of the cybersecurity challenges. For example, both supervised and unsupervised learning algorithms are used for problems like phishing detection, anomaly detection, threat analysis, malware discovery, biometric recognition, and many others. See Dua and Du (2011) for a more comprehensive list of areas of cybersecurity where machine learning is being used.

In this work, we are going to focus on supervised and unsupervised learning algorithms, which can be further split into two classes each:

- Supervised learning
  - *Classification*: Algorithms that take labelled data and learn patterns that can be used to predict a categorical output variable.

  - *Regression*: Algorithms that take labelled data and learn patterns that can be used to predict a continuous output variable.

- Unsupervised learning
  - *Clustering*: Algorithms covering a range of techniques that try to identify groups within a dataset. A cluster is composed of points that are more similar to each other. Clustering can be considered similar to classification in that we are trying to assign a discrete value to each datapoint. However, unlike classification, we do not have labels to learn patterns in the data that separate the classes. Instead, we use clustering when we do not have prior knowledge of the class membership.
  - *Dimensionality reduction*: Algorithms that take high-dimensional unlabelled data and learn a lower-dimensional representation. They are usually used as an exploratory technique and as preprocessing step in the model building pipeline.

Chapter 1

Cluster Analysis

In this chapter, we will tackle the task of characterising the attacks that were carried out against a *Cowrie* honeypot by using clustering algorithms. We make use of unsupervised learning algorithms because we are dealing with unlabelled data: the information we have about the attacks are the logs gathered by the honeypot, which contain a set of characteristics of each attack. By using clustering algorithms, we analyse these characteristics to find patterns in the data with the objective of gaining insights into the attacks. This is relevant to cybersecurity experts because the results can help them draw conclusions about the attacks, such as their origin or the intentions of the attacker.

Cluster analysis has the objective of partitioning the data into subsets, or clusters, in such a way that the datapoints within each cluster are more closely related to each other than to those assigned to the other clusters.

Although there are many clustering algorithm classifications, these are the four most popular:

- *Partitional algorithms*: These algorithms find groups present in the data by optimising an objective function and iteratively improving the quality of the partitions.
- *Hierarchical algorithms*: Algorithms that find clusters by creating a binary tree hierarchy called the *dendogram*. The strategies for hierarchical clustering generally fall into two types: agglomerative clustering and divisive clustering.
- *Density-based clustering algorithm*: These are algorithms that find groups in the data based on the idea that a cluster in a data space is a contiguous region of high density, separated from other clusters by contiguous regions of low density.
- *Probabilistic models for clustering*: These algorithms are based on the assumptions that the data are generated by a mixture of underlying probability functions (e.g., Gaussian or Poisson distributions).

In this work, we use two partitional algorithms, the k-means algorithm (Forgy, 1965; MacQueen, 1967) and the k-prototypes algorithm (Huang, 1997a), and a density-based clustering algorithm, density-based spatial clustering of applications with noise, or DBSCAN (Ester et al., 1996). Each has its advantages and shortcomings depending on several factors, as we will see in the following sections.

The theoretical background of the algorithms is introduced in Section 1.1. In Section 1.2 we use the algorithms on real data, show the results and give the conclusions of the analyses.

1.1 Overview of clustering algorithms

1.1.1 K-Means Clustering

The *k-means* algorithm is one of the most popular and studied clustering algorithms due to its simplicity, ease of implementation and efficiency. In some applications, the algorithm can be used as part of a pre-processing stage to label the data, which can later be used to build classification models. In our case, however, we will use the algorithm in an unsupervised context only to find groups in the honeypot data in order to find patterns and draw conclusions from them.

K-means is a partitional clustering algorithm, meaning it finds partitions in the data and iteratively improves said partitions. Such partitions divide the data space into a structure known as *Voronoi* cells (see Figure 1.1). The algorithm starts by choosing k points as the initial centroids. Each datapoint is assigned to the closest centroid based on the proximity measure that has been chosen. New centroids are calculated based on the new datapoints assignments. These two steps are iteratively repeated until no new datapoints assignments are made, or another convergence criterion is met.

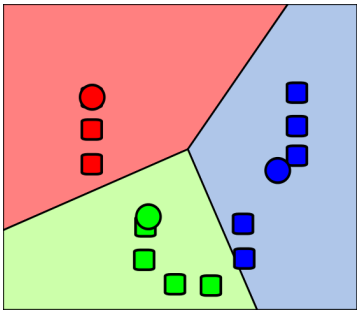


Figure 1.1: In this image k-means is applied to data in two dimensions. We can see the resulting Voronoi partitions, each in a different colour, that divide the space to form the clusters. Source: Wikipedia

Algorithm 1.1: K-means clustering algorithm

1. Select K points as initial centroids, $c_1, c_2, \ldots, c_k$.
2. Form K clusters by assigning each point to its closest centroid.
3. Recalculate the centroid of each cluster.
4. Steps 2 and 3 are repeated until the assignments do not change.

To calculate which centroid is closest to a given datapoint, a proximity measure must be used. There are a number of proximity measures that can be used for the k-means algorithm. The chosen proximity measure can greatly affect the centroid assignment and therefore change the final solution.

The most used measures include the Manhattan distance (L_1 norm), Euclidean distance (L_2 norm), and the Cosine similarity; for k-means clustering, the Euclidean distance is the most popular choice. The Euclidean distance between two P-dimensional vectors, x_i and $x_{i'}$, is defined as

$$d(x_i, x_{i'}) = \sum_{j=1}^{P} (x_{ij} - x_{i'j})^2 = \|x_i - x_{i'}\|^2$$

where x_{ij} is the value of the jth element for the ith vector and $j = 1, \ldots, P$.

The k-means algorithm can be viewed as an optimisation problem. Given a dataset $D = \{x_1, x_2, \ldots, x_N\}$ of N data points, we denote the clusters obtained after applying k-means algorithm as $C = \{C_1, C_2, \ldots, C_K\}$. The algorithm works by minimising an objective function, which in this case is the sum of squared errors, or SSE, defined as

$$SSE(C) = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - c_k\|^2$$

$$c_k = \frac{\sum\limits_{x_i \in C_k} x_i}{|C_k|}$$

where c_k is the centroid vector associated with the kth cluster C_k, and $|C_k|$ is the number of data points in the cluster C_k. In each iteration, the k-means algorithm aims to minimise the SSE for the given centroids in the assignment and update steps. Figure 1.2 shows a visual representation of the algorithm with toy data. The SSE decreases monotonically with each iteration. Therefore, the algorithm will converge to a local minimum. Even though convergence is assured, the algorithm may find a suboptimal local minimum; hence, it is important to start the algorithm with many initialisations for the starting centroids, and then choose the solution having the smallest value of the objective function. Algorithm 1.1 shows an outline of the k-means algorithm.

Initialisation Methods

The simplest method of initialising the K centroids is by selecting the seeds at random, as proposed by MacQueen (1967). Other initialisation methods exist which have been proven to improve the clustering performance. Arthur and Vassilvitskii (2007) proposed the k-means++ algorithm, which follows a probabilistic approach, where the centroids are selected using a weighted probability score. Another method is the one proposed by Hartigan and Wong (1979) based on the concept of nearest neighbour density, which finds candidate points that have a large number of points within their surrounding sphere. An additional popular method was proposed by Milligan (1981), which uses the results from applying agglomerative hierarchical clustering to choose the initial centroids. A more thorough comparison of initialisation methods for k-means can be found in Celebi et al. (2013).

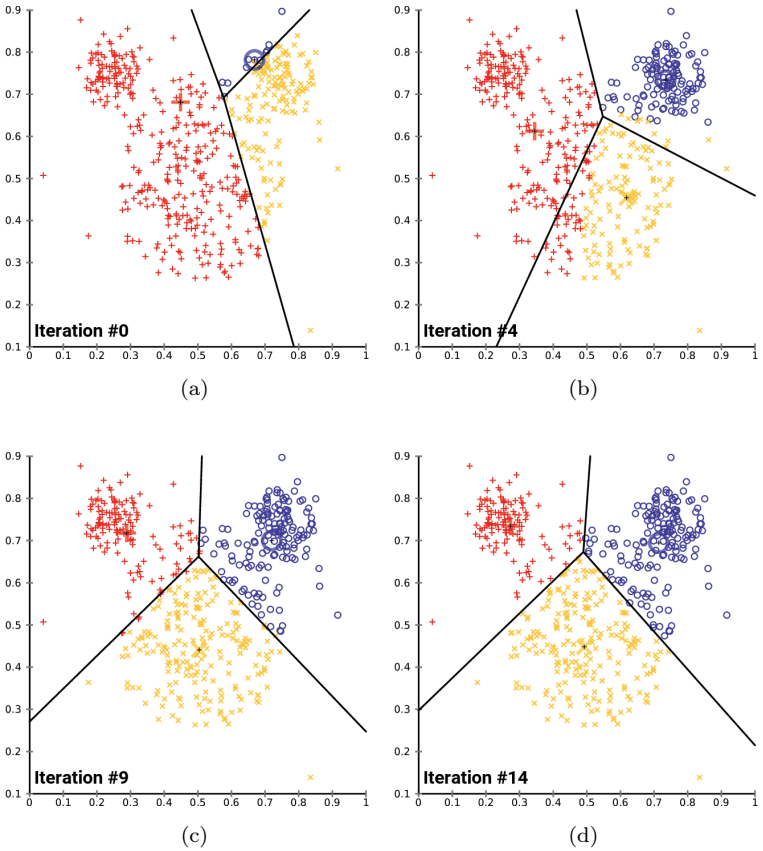


Figure 1.2: Figure showing four of 14 iterations of the k-means algorithm for a particular dataset, for $K = 3$. The algorithm starts at iteration 0 with a random initialisation of the centroids. At iteration 9, the algorithm has found nearly optimal centroids, making only small changes in the subsequent iterations. By iteration 14, the algorithm has already converged. Source: Wikipedia

Estimation of the number of clusters K

A major challenge the k-means algorithm presents is the estimation of the correct number of clusters. In some cases, the number of clusters can be specified by the context of the problem. For example, one could have the goal of segmenting the data into three parts. Other times, an expert in the domain of the problem may indicate the number of clusters in the data. In our case, however, we do not know the number of clusters and need to estimate it. Many methods have been proposed for solving the problem of selecting the number of clusters, most of them relying on the calculation of *internal cluster metrics*, which are statistics that aim to quantify the quality of a clustering result.

These metrics are called internal because they are calculated from the results of applying the clustering algorithm to the data, as opposed to the *external cluster metrics* which compare the results to external labels, or the true known clusters. The metric we will use is the silhouette coefficient (Kaufman and Rousseeuw, 1990).

The *silhouette coefficient*, or silhouette score, is an internal clustering metric used to calculate the goodness of a clustering algorithm. Its value ranges from -1 to 1, where values closer to 1 indicate the clustering results are adequate. This metric considers both the distances between the points within each cluster (intra-distance) and the distance between the clusters (inter-distance). For a given data point x_i belonging to cluster C_i, let

$$a_i = \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(x_i, x_j)$$

be the average of the distances to all points in the same cluster, where $|C_i|$ is the number of points belonging to the ith cluster, and $d(x_i, x_j)$ is the distance between data points x_i and x_j that belong to cluster C_i. The value of a_i can be interpreted as a measure of how well the data point x_i was assigned to its cluster. We now define the mean dissimilarity of x_i to some cluster C_k as

$$b_i = \min_{k \neq i} \frac{1}{|C_k|} \sum_{j \in C_k} d(x_i, x_j)$$

to be the smallest mean distance from x_i to all points in any other cluster C_k, where $i \neq k$. Using a_i and b_i the estimated silhouette coefficient of a point is

$$S_i = \frac{b_i - a_i}{\max(a_i, b_i)}.$$

To evaluate the quality of a cluster one can compute the average silhouette coefficient of all points as

$$S = \frac{\sum_{i=1}^{N} \frac{b_i - a_i}{\max(a_i, b_i)}}{N}$$

where N is the total number of data points.

1.1.2 K-prototypes Clustering

The k-prototypes algorithm is a clustering algorithm that can handle mixed data types, that is, a mixture of categorical and numerical variables. Under the hood, it works as an integration of the k-means and k-modes[1] algorithms. Because of its ability to handle categorical and numerical variables, it is a very useful algorithm since frequently encountered datasets contain mixed-type data.

The k-prototypes algorithm works by using a distance metric that weighs the contribution of the numerical against the categorical attributes. The steps the algorithm takes in order to find clusters in

[1]The k-modes algorithm (Huang, 1997b) extends the k-means algorithm to cluster categorical data by using a simple matching dissimilarity measure for categorical objects, modes instead of means for clusters, and a frequency-based method to update modes to minimise the clustering cost function.

Algorithm 1.2: K-prototypes clustering algorithm

1. Select K points as initial centroids, $c_1, c_2, \ldots, c_k$.
2. Form K clusters by assigning each point to its closest centroid.
3. Recalculate the centroid of each cluster.
4. Steps 2 and 3 are repeated until the assignments do not change.

the data are shown in Algorithm 1.2, which are practically the same as the steps of k-means, with the difference that k-prototypes uses a different dissimilarity measure to calculate distances which includes categorical variables as well. The dissimilarity between two mixed-type data points x_i and x_j, each consisting of P numerical predictors and $M - P$ categorical predictors, can be measured by

$$d(x_i, x_{i'}) = \sum_{j=1}^{P}(x_{ij} - x_{i'j})^2 + \gamma \sum_{j=P+1}^{M} \delta(x_{ij}, x_{i'j}), \tag{1.1}$$

being the first term is the square Euclidean distance measure on the numerical attributes. The second term is the dissimilarity measure of the categorical attributes and is defined as

$$\delta(x_i, x_{i'}) = \begin{cases} 0, & x_i = x_{i'} \\ 1, & x_i \neq x_{i'} \end{cases}.$$

The second term of Equation 1.1 is the sum of mismatches in the categorical attributes of the two datapoints; the smaller the number, the more similar the two data points are. The weight γ is used to adjust the importance of the dissimilarity of the categorical attributes. We can analyse the effect of weight γ using Figure 1.3. The figure shows a set of objects described by a single categorical and two numeric attributes. The levels of the categorical variable are defined by the shapes of the objects. The value of the numeric attribute is represented by each axis. If $\gamma = 0$, the algorithm does not take into account the categorical attribute and will separate the objects by their numerical attributes, resulting in two clusters divided by the vertical dashed line. If $\gamma > 0$, objects b and c may change clusters. Object b might change to the cluster in the right, while c might get assigned to the cluster in the left. Object d might become uncertain; depending on whether the value of γ is large or small, it will get assigned to either cluster. Objects a and e are most likely to remain on the left and right clusters, respectively.

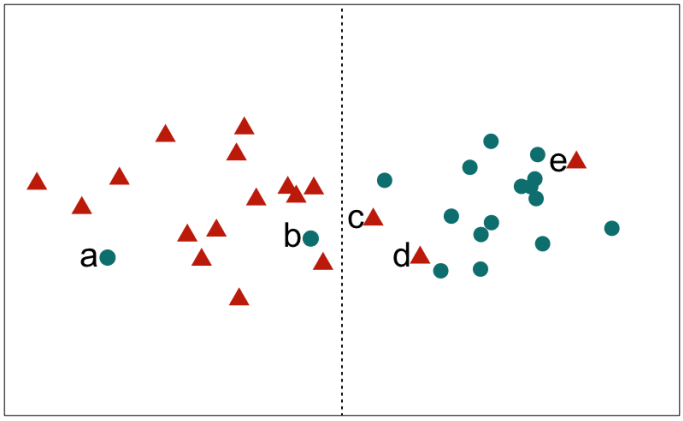


Figure 1.3: Influence of γ in clustering.

In Huang (1997a), the author states that the choice of γ depends on the distributions of the numeric attributes. The value of γ is related to the average standard deviations of numeric attributes in a cluster C_k. However, because the clusters are not known, it is not possible to calculate this. Instead, the overall average standard deviation of the numeric attributes can be used.

In the same way as the k-means algorithm, the k-prototypes also produces local optimal solutions, which are affected by the initial cluster prototypes. To avoid being trapped at a local optimum, one can execute the algorithm several times with different initialisations. K-prototypes algorithm consists of three main steps: initialisation of centroids, cluster assignment of datapoints, and centroids update. The initial selection of the centroids is usually made by selecting at random K objects.

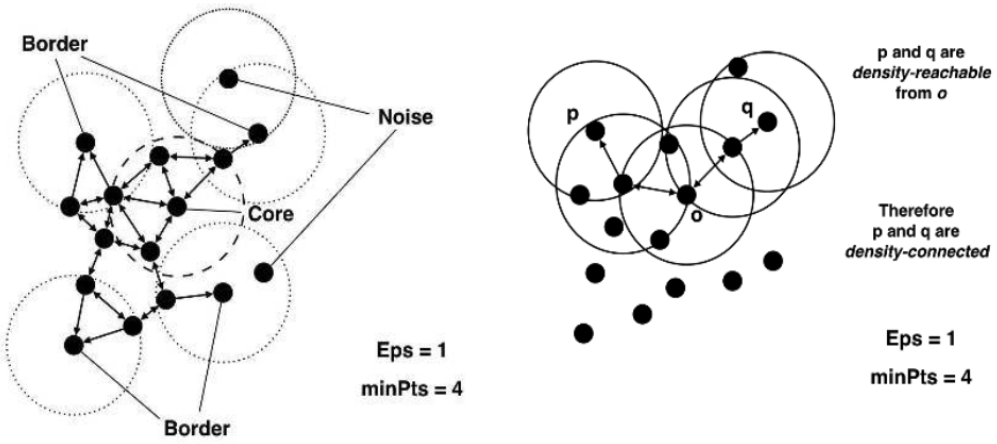
1.1.3 DBSCAN Clustering

Density-based algorithms aim to achieve the same as partitional algorithms: partition the data into finite groups that reveal structure in the data. Unlike the k-means and k-prototypes algorithms, which work by measuring distances between each datapoint and the calculated centroids, density-based clustering algorithms use *density* to assign cluster membership. Density, in this context, could be defined as the number of data points per unit volume of the feature space. Areas of the feature space containing many data points packed closely together can be said to have high density, while areas of the feature space containing few or no data points can be said to have low density. We can expect a cluster to be formed with points within a region of high density, separated from other clusters by regions of low density.

Density-based spatial clustering of applications with noise, or DBSCAN, estimates the density by counting the number of points in a fixed radius neighbourhood. The algorithm starts by selecting a point in the data and searching for other points within a search radius. This radius is determined by the *eps* parameter. So *eps* represents how far away from each data point (in an n-dimensional sphere) the algorithm will search for other data points around a given data point. The parameter *eps* is expressed in units of the feature space, and the distance metric it uses is the Euclidean distance by default. Larger values of *eps* mean the algorithm will search further away from each data point. The *minPts* parameter specifies the minimum number of points (cases) that a cluster must have in order for it to be a cluster. The *minPts* parameter is therefore an integer.

The DBSCAN algorithm makes the distinction between three different types of points (see Figure 1.4):

- *Core point*: A point p is a core point if the neighbourhood of radius *eps* contains at least $minPts$ points.
- *Border point*: A point that is reachable from a core point p but is not itself a core point.
- *Noise points* or *outliers*: All points that do not fall in any of the other categories, i.e., points not reachable from any other point.



(a) (b)

Figure 1.4: Concepts used the DBSCAN family of algorithms. 1.4a shows examples for the three point classes, core, border, and noise points, 1.4b illustrates the concept of density-reachability and density-connectivity. Source: Ester et al. (1996)

A more formal definition of the algorithm is the following: let D be a set of datapoints. The *eps*-neighbourhood of a point p, denoted by $N_\varepsilon(p)$, is defined by $N_\varepsilon(p) = \{q \in D \,|\, dist(p,q) \leq eps\}$. A point p is *directly density-reachable* from a point q with respect to *eps* and $minPts$ if

(1) $p \in N_\varepsilon(q)$ and
(2) $|N_\varepsilon(q)| \geq minPts$,

and point p is *density-reachable* from a point q with respect to *eps* and $minPts$ if there is a chain of points $p_1, \ldots, p_n$, $p_1 = q$, $p_n = p$ such that $p_i + 1$ is directly density-reachable from p_i. A point p is *density-connected* to a point q, given *eps* and $minPts$, if there is a point o such that both p and q are density-reachable from o with respect to *eps* and $minPts$.

A cluster in DBSCAN is a maximal set of density-connected points. Then, a cluster C, given *eps* and $minPts$, is a nonempty subset of D that satisfies the following conditions:

(1) $\forall p, q$ if $p \in C$ an q is density-reachable from p, then $q \in C$.

(2) $\forall p, q \in C : p$ is density-connected to q with respect to eps and $minPts$.

Let $C_1, \ldots, C_K$ be the clusters formed by the algorithm, given eps and $minPts$. Then the noise points are defined as the set of points in D that do not belong to any cluster. Algorithm 1.3 shows the steps carried out by DBSCAN.

Algorithm 1.3: Abstract DBSCAN algorithm (Schubert et al., 2017)

1. Compute the neighbours of each point and identify core points.
2. Join neighbouring core points into clusters.
3. **foreach** non-core point **do**
   (a) Add to a neighbouring core point if possible.
   (b) Otherwise, add to noise.

Estimation of the parameters *eps* and *minPts*

The DBSCAN algorithm has two parameters that need to be specified, eps and $minPts$. As it was mentioned before, eps represents how far away from each point the algorithm is going to look for other points. Usually, this value is selected by the following heuristic: the distance from each point to its kth-nearest neighbour is computed, and then the points are ordered in a plot based on this distance (Ester et al., 1996). An example of such plot can be seen in Figure 1.5. In data with regions of high and low density, this tends to produce a plot containing a "knee". The author of the algorithm suggests that the optimal value of eps is going to be close to that "knee" (Ester et al., 1996). Because each core point in DBSCAN has $minPts$ in its eps-neighbourhood, choosing a value of eps at the "knee" means selecting a distance that has high-density regions of core points (Ester et al., 1996). Selecting values of eps that are too small will result in the formation of many small clusters and possibly a large part of the data not being clustered; if the value is too high, clusters will merge together, creating few, large clusters. The value for the parameter in the kth-nearest neighbour, k, is usually set as $k = minPts - 1$ (Schubert et al., 2017).

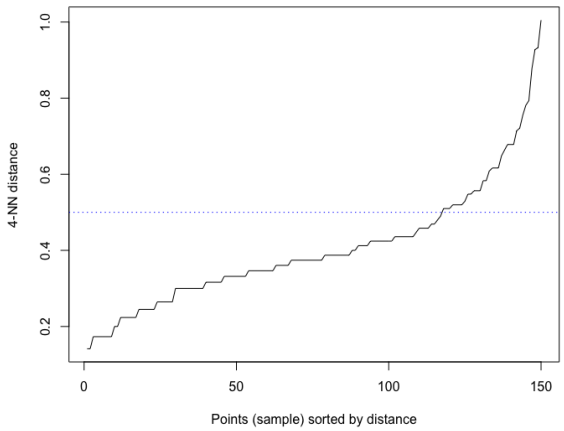


Figure 1.5: Example of k-distance graph using the *iris* dataset, when the parameter k is set to 4. The "knee" is around 0.5, as shown by the dotted line in blue.

Several heuristics have been developed for choosing an appropriate value for $minPts$. For 2-dimensional data, setting $minPts$ to 4 has been proven to work well since, according to the author, the k-distance graphs for $k \geq 4$ do not significantly differ from the 4-distance graph (Ester et al., 1996). Another heuristic is setting $minPts$ to 2 times the number of dimensions in the data (Sander et al., 1998). If the dataset is very large, is known to be very noisy, and has duplicates, larger values are preferred (Schubert et al., 2017). Furthermore, someone that has domain knowledge can help select the value of $minPts$; the value of the parameter could be set as the approximate number of elements that are expected to be in a cluster. This is due to the fact that $minPts$ defines the minimum number of points in a cluster.

There are a few advantages of using density-based algorithms, such as DBSCAN, over partitional ones. Here are a few:

- The major advantage is that we do not have to specify the number of clusters before the analysis, as opposed to k-means and k-prototypes.
- They do not make any assumptions about the shape of the clusters, therefore can find clusters of varying and complex shapes.
- They are able to detect cases that do not fall within regions of high density areas, and put them into a *noise* cluster, making it robust to outliers.

Even though it has very desirable properties, it also has some disadvantages:

- The parameters $minPts$ and eps need to be specified.
- It suffers from the *curse of dimensionality* when dealing with high-dimensional data.
- DBSCAN does not group data well if the density of the cluster varies greatly.

UMAP dimensionality reduction algorithm

In this work, we made use of a dimensionality reduction algorithm in combination with the DBSCAN algorithm as we explored new ideas to tackle the data characterisation task. The algorithm we used is called *uniform manifold approximation and projection for dimension reduction*, or UMAP (McInnes et al., 2018). Even though the algorithm is relatively new, it already has been used successfully across multiple fields as a way to prepare the data in a step prior to using clustering algorithms. See Allaoui et al. (2020); Stankowicz and Kuzdeba (2021) for some examples.

UMAP is a nonlinear dimensionality reduction algorithm that aims to preserve local neighbourhoods in the data. It is based on manifold learning techniques and ideas from topological data analysis. The algorithm consists of two phases: the first phase constructs a fuzzy topological representation of the data. For each individual datapoint, the algorithm identifies a preset number of nearest neighbours. Then it constructs a weighted graph, representing the distance between the nearest neighbours with weights, where the neighbours that are closer together are weighted more heavily. The second phase optimises the low dimensional representation of the data to represent the local neighbourhoods as much as possible.

In the graph construction phase, UMAP uses the k-nearest neighbours algorithm, or k-NN, to construct a weighted k-NN graph from the data where each vertex represents a datapoint, and edge weights represent the likelihood that two points are connected. Let $D = \{x_1, \ldots, x_N\}$ be a dataset of N points, then for each point x_i, the set of points in its local neighbourhood is computed using a distance metric d. The default distance metric used by UMAP is the Euclidean distance, but other metrics can be used. Let $\mathcal{M}_i = \{x_i^{(1)}, \ldots, x_i^{(k)}\}$ be the neighbourhood of x_i with k neighbours. Then for each x_i, we have ρ_i defined as

$$\rho_i = \min\{d(x_i, x_j) | d(x_i, x_j) > 0, x_j \in \mathcal{M}_i\}$$

which is a connectivity constraint that datapoints are assumed to be locally connected. This ensures that x_i is connected to at least one other datapoint with a weight of 1. Let $G = (V, E, w)$ be the weighted graph where the vertices V are the set of datapoints D. Then the weight function which represents the fuzzy connection between points is given by

$$w_h(x_i, x_j) = \exp\left(\frac{-\max(0, d(x_i, x_j) - \rho_i)}{\sigma_i}\right)$$

where σ_i is a normalisation factor for the metric d local to the points x_i, defined as

$$\sigma = \sum_{i=1}^{k} w_{h_i}.$$

Once the high-dimensional graph is constructed, UMAP constructs and optimises the layout for the low-dimensional representation of the data in $\mathbb{R}^m$. To accomplish this, Laplacian eigenmaps are used. The ith datapoint is represented as a vector of $\mathbb{R}^m$ coordinates $(f_1(i), \ldots, f_m(i))$ where $f_0, \ldots, f_m$ are the eigenvectors of the Laplacian associated with G. The algorithm then constructs a low-dimensional

graph G' from elements of $\mathbb{R}^m$, using the same approach it did for constructing G except now it uses other weights which are defined between two points $z_i, z_j \in \mathbb{R}^m$ as

$$w_l(z_i, z_j) = \frac{1}{1 + a(||z_i - z_j||_2^2)^b}$$

where parameters a and b are chosen such that the weight function w_l gives a smooth approximation, by fitting non-linear squares, of Ψ defined as

$$\Psi(z_i, z_j) = \begin{cases} 1 & \text{if } ||x_i - z_j|| \leq min_dist \\ \exp(-(||z_i - z_j||_2 - min_dist)) & otherwise \end{cases}$$

where min_dist is a hyperparameter of the UMAP algorithm. This approximation is necessary to allow for the cross-entropy between G and G' to be differentiable. Finally, UMAP optimises the low-dimensional representation by minimising the cross entropy between G and G' by using the stochastic gradient descent algorithm.

Because the UMAP algorithm is stochastic, each time the algorithm is executed the results will be different, even if the same hyperparameters are selected. Additionally, since the choice of hyperparameters is so important, it can be very useful to run the projection multiple times with various hyperparameters.

UMAP hyperparameters

The UMAP algorithm takes many hyperparameters. These are the most relevant:

- $n_neighbours$, the number of neighbours to consider when approximating the local metric
- min_dist, the desired separation between close points in the embedding space
- d, the target embedding dimension

The $n_neighbours$ hyperparameters defines the number of neighbours to be used by k-NN algorithm when constructing the high dimensional graph. This hyperparameters is how UMAP balances local versus global structure in the data. Low values of $n_neighbours$ will make the algorithm to concentrate on very local structure, while large values force UMAP to look at larger neighbourhoods of each datapoint, losing fine detail for the sake of capturing a broader structure of the data. On the other hand, the min_dist hyperparameter controls how tightly UMAP packs datapoints together. It provides the minimum distance apart that points are allowed to be in the low dimensional representation. Thus, smaller values of min_dist will result in clumpier low-dimensional embeddings; larger values will make the points in the embedding be more spread out. Tha parameter d is the desired dimension of the low-dimension representation.

1.2 Application to honeypot data

Having introduced the theoretical background of clustering analysis, we are going to proceed to the practical analyses using honeypot data. We begin this section by describing the data that will be used throughout the analyses. Then, we will show the results from applying each of the clustering techniques described in the previous section. Finally, we finish this section with a discussion of the results found by each of the algorithms.

1.2.1 Description of the data

For the clustering analysis task, we used data collected by a *Cowrie* honeypot between the dates of April 3 and May 17, 2017. The data originally consisted of 44 log files in *JSON* format. These files were parsed by a member of the cybersecurity team at Gradiant in a previous step, not included in this work, to extract relevant variables about the attacks made to the honeypot. The data has been structured into a data frame to make it easier to manipulate in R.

The dataset used in this section consists of 11 variables (10 numerical and 1 categorical) and 9622 observations. A description of the variables is shown in Table 1.1. As a side note, we should point out that the number of variables extracted from the logs is greater than the ones used here. However, we selected the variables we believed to be the most relevant and helpful for the analysis.

We considered all the datapoints for which the variable $n_tries_logins > 0$, i.e., datapoints with at least 1 login attempt. We do this to filter out *ping*-only attempts (datapoints where a connection was made but there were no attempts to log in), which can be considered harmless scans of the system. Additionally, we only considered the cases for which the attack had a successful login to the honeypot, as we are interested in analysing the behaviour of the attacker once they have successfully logged into the system.

1.2.2 K-means clustering results

In order to find clusters in the data, we made use of the k-means clustering algorithm described in Section 1.1.1. We used the *MLR* package because it provides a framework that integrates all the steps that were performed in the analyses. It is important to note that we only used the first 10 numerical variables listed in Table 1.1, and did not include the categorical variable *sshversion* in this analysis because, as mentioned earlier, the k-means algorithm cannot handle categorical variables.

Data preprocessing

The first step in the clustering analysis consisted in performing some data preprocessing. The variables in the data are not in the same scale, therefore their ranges and magnitude vary greatly. The k-means algorithm is affected by this characteristic because it relies on the square of the Euclidean distance, making it highly dependent on the measurement scales of the variables. Since the features have different scales, the algorithm will give more importance to the variables with higher magnitudes, changing the results of the analysis.

Name of the variable	Type	Description
n_tries_logins	numeric	Number of login attempts
words_user	numeric	Mean number of characters of each username used during the login of the same session
words_pass	numeric	Mean number of characters of each password used during the login of the same session
duration_time_login	numeric	Time, in seconds, of the duration of the login of a session
n_connections	numeric	Number of outgoing connections during a session
duration_inside	numeric	Time, in seconds, of the duration of a session
nfiles_down	numeric	Number of downloaded files during a session
nfiles_create	numeric	Number of created files during a session
ncommands	numeric	Number of executed commands during a session
score_ip	numeric	Score of IP address of the attacker. Low values indicate low risk IPs, while large values indicate high risk IPs. The values go from 0 to 100
sshversion	categorical	SSH protocol version used by the attacker

Table 1.1: Name, type and description of the variables used in the analyses.

There are many approaches to solve this issue. One of them is to standardise the variables to have a 0 mean and standard deviation of 1. Another popular approach, which is the one we used, is to rescale the variables by dividing them by their range, as this technique has been found to perform well in clustering analysis (Milligan and Cooper, 1988).

Tuning k-means algorithm

The next step in the analysis is to select the number of clusters used by the k-means algorithm, k. Because the real number of clusters in the data is unknown, we need an approach to find a reasonable value for this hyperparameter. However, we need to be careful not to overfit (or underfit) our model when doing this. Overfitting is said to happen when the model does not generalise well from training data to unseen data. Overfitting occurs when the parameters of a model are chosen so that the model becomes too flexible and captures the noise in the data. On the other hand, underfitting happens when the model is too simple and is not able to capture the variability in the data. Overfitting/underfitting is undesirable because the fitted model will not yield accurate estimates or results based on new observations that were not part of the data used for training. In clustering analysis, if the model underfits the data, it will fail to identify and separate real clusters in the data. Alternatively, if the model overfits the data, it will create much more clusters that do not exist.

The approach we used to solve this is K-fold cross validation. K-fold[2] cross validation splits the data into K folds of equal size; then for each fold $k \in 1, \ldots, K$, we train the model on all the folds but the kth, and then use the kth fold, as sketched in Figure 1.6, to measure the performance of the model. This procedure is repeated K times so that we obtain K performance estimates. These

[2]Do not confuse the K in K-fold cross-validation with the hyperparameter k which denotes the number clusters in k-means algorithm.

estimates are then averaged to obtain a single performance estimate. The metric we used to measure the performance is the Silhouette score introduced in Section 1.1.1. Typical values for the number of folds are $K = 5$ and $K = 10$. In our analysis, we used $K = 10$ folds, which is a good trade-off between average performance estimation and the computational cost of calculating K-fold cross validation (see Hastie et al. (2001) for more details).

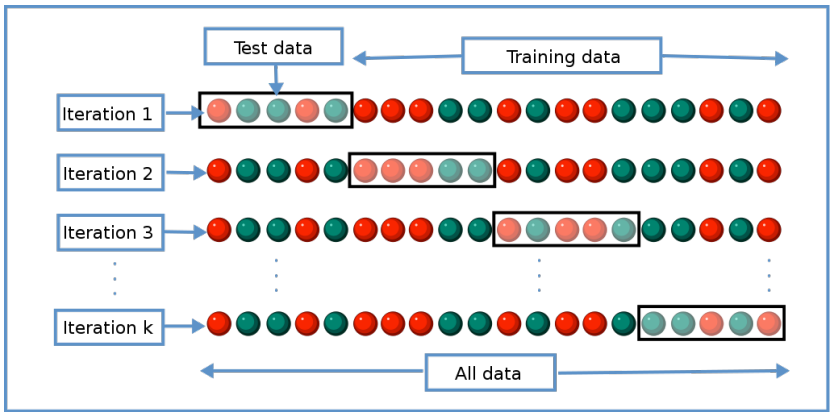


Figure 1.6: Image representation of how K-fold cross validation works. The data is divided into K-folds. For each iteration, the kth fold is treated as the test set and the rest $K - 1$ folds as the training set. Source: https://en.wikipedia.org/wiki/File:K-fold_cross_validation_EN.svg

We began the analysis by defining the discrete search space considered for the number of clusters, k. We want to find the optimal number of clusters in the range from 2 to 10 (in a preliminary analysis, we found that increasing the number of clusters beyond 10 did not improve the Silhouette score). For each number of clusters considered, the data was split into 10 folds, and the k-means model was trained on the training set of each fold. Using this trained model, the cases in each test set were assigned to their respective clusters, and the Silhouette score was calculated on the resulting clusters. Finally, the average Silhouette metric of all the 10 folds for each value of k is calculated. The results of this computation can be seen in Table 1.2. A graph of the values is shown in Figure 1.7.

Num. of centers (k)	Mean test Silhouette score
2	0.6739
3	0.6473
4	0.7694
5	0.7710
6	0.7695
7	0.7749
8	0.7816*
9	0.7224
10	0.6952

Table 1.2: Silhouette test means scores obtained from applying 10-fold cross validation to the data.

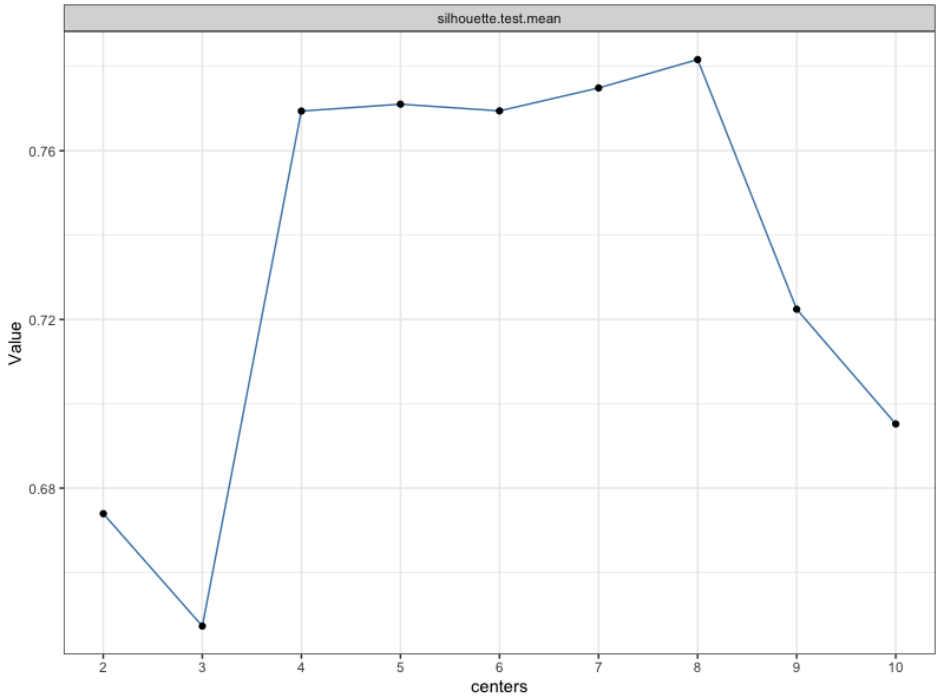


Figure 1.7: Plot of the mean test Silhouette scores calculated from the 10-fold cross validation approach, used for tuning the number of clusters in the k-means algorithm.

From the values calculated, we see that the optimal number of clusters obtained is $k = 8$, with a Silhouette score of 0.7816, followed closely by $k = 7$ with a score of 0.7749. Note that the difference between them is < 0.1; therefore, $k = 7$ could potentially be a good value for the number of clusters. We also see that the values of the Silhouette score starting from $k = 4$ to 8, are high; further increasing the number of clusters above 8 does not improve the Silhouette score.

Training the k-means model

Once we tuned the k-means algorithm, we trained the model using the optimal value for the number of clusters that was found, $k = 8$. Table 1.3 shows the number of datapoints that were assigned to each cluster of the trained model. The cluster with the largest percentage of data is cluster 8, containing 3444 datapoints (44.12% of the data). The second largest cluster is number 2 with 1907 datapoints. The clusters 1, 5 and 6 are the three smallest clusters, number 6 being the smallest of all containing 119 datapoints. Table 1.4 shows some metrics calculated for the variables of all the datapoints contained within each cluster.

Cluster number	1	2	3	4	5	6	7	8
Number of datapoints	263	1907	717	418	157	119	781	3444
Percentage of data	3.37%	24.43%	9.19%	5.35%	2.01%	1.52%	10.01%	44.12%

Table 1.3: Number of datapoints in each cluster. The last row shows the percentage of all the data contained in each cluster.

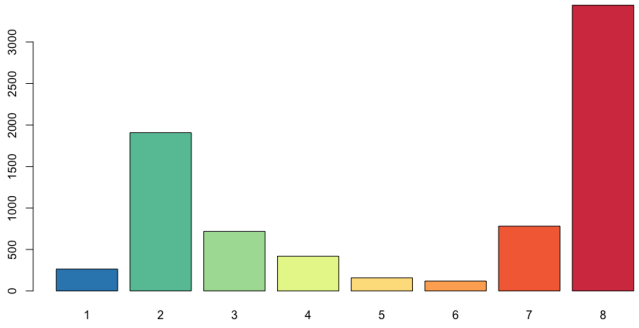


Figure 1.8: Plot showing the number of datapoints in each cluster.

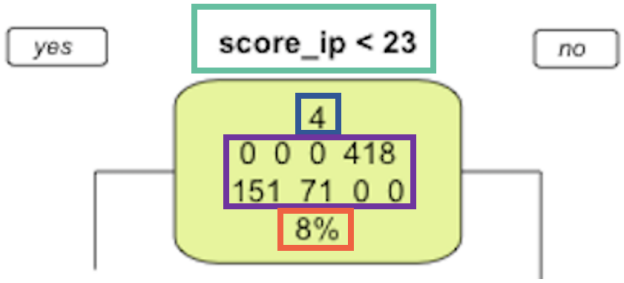


Figure 1.9: Description of the parts of a single node in the decision tree. The green rectangle surrounds the name and the value of the variable that was chosen for the split in any given node. In this particular node, the variable is $score_ip$ and the threshold value of the split is 24 (datapoints with value of $score_ip < 24$ are assigned to the left child node of the current node; alternatively, datapoints with $score_ip \geq 24$ are assigned to the right child node). The number 4, surrounded by the blue square, indicates the cluster number to which the majority of datapoints in the node are assigned to. The following 8 numbers, enclosed by the purple rectangle, indicate the exact number of datapoints belonging to each cluster. For example, the data in the current node is composed of 418 datapoints of cluster 4, 151 datapoints of cluster 5, and 71 datapoints of cluster 6. There are no datapoints of clusters 1, 2, 3, 7, and 8 in the node. Finally, the number in the orange rectangle is the percentage of datapoints, with respect to all the data, that is contained in the node. In this case, the node contains 8% of all the datapoints.

To help us interpret the results obtained by the k-means algorithm, we decided to use the decision tree algorithm for classification given its high interpretability. A decision tree is a supervised machine learning algorithm that constructs a tree-like structure by making recursive binary splits of the data according to certain cutoff values in the variables. The goal of the decision tree for classification is to find the best splits that increase the purity or homogeneity of the data in the sub-nodes. The idea is to use the decision paths found by the decision tree algorithm to gain insights of the datapoints assigned to each cluster. With this in mind, we trained a decision tree model using the clusters found by the k-means algorithm as the target variables, or labels. The resulting decision tree is depicted in Figure 1.10. A detailed description of the information that appears in a node of the tree is shown in Figure 1.9. On top of each node we can see the variable and its cutoff value chosen by the decision tree for the split. Inside each node, the number at the top indicates which cluster the node belongs to: this is defined by counting which is the largest cluster in the node. The following 8 numbers show the amount of datapoints assigned to each of the 8 clusters, given the data contained in the node. The percentage at the bottom reflects the percentage of data contained in the node.

From a quick analysis of the decision tree shown in Figure 1.10, we can already tell many things. Out of the 10 variables that were used in the training of the k-means model, the decision tree only shows decisions for 5 of them: $n_connections$, $ncommands$, $score_ip$, $duration_inside$, and $nfiles_down$. The variables $n_connections$, $ncommands$, and $duration_inside$ appear two times in the tree, suggesting the decision tree found them relevant for the splits. The first split of the tree, the one at the root node, was made on $n_connections \geq 0.5$. This divides the data into the datapoints where 1 or more outgoing connections were made (left branch), and datapoints with 0 outgoing connections (right branch). By looking at the leaf nodes (the ones at the bottom of the decision treee), we see that clusters 1, 4 and 6 achieved 100% purity, meaning there are no datapoints belonging to other clusters in their nodes.

If we follow the path from the root node of the decision tree down to the leaf node of cluster 1 (in blue), we can see it is characterised by datapoints with 0 outgoing connections, having executed at least 5 commands, and downloaded more than 4 files. We see in Table 1.4, that cluster 1 contains the datapoints which had more downloaded files ($nfiles_down$ column) compared to the other clusters, with a mean of 6.3498 downloaded files. It also contains points that executed many commands ($ncommands$), which is to be expected, as the attacker needs to do so in order to download the files.

Cluster 2, the second largest cluster with 24.43% of the datapoints, is characterised by datapoints with a number of outgoing connections $n_connections$ between 1 and 2; if we look at the Table 1.4, we see that the mean of $n_connections$ is 1.0058, and its median is 1, suggesting this cluster is mostly composed of points which made a single outgoing connection. The datapoints in this cluster seem to belong to "tunnel connection" behaviour, where the attacker used the honeypot to forward 1 connection to another system creating a tunnel.

Cluster 3 is very similar to 1: it contains datapoints with 0 outgoing connections and datapoints where at least 5 commands were executed. However, the difference is that cluster 3 contains data having a smaller number of downloaded files $nfiles_down$ (3 or less downloaded files).

Datapoints in cluster 4 are characterised by having a large number of outgoing connections compared to other clusters, except for clusters 5 and 6. Indeed, from the decision tree we can see that the leaf nodes belonging to these clusters (4, 5, and 6) fall to the right of the node with the split of $n_connections > 2.5$. From the summary statistics table, we can also see that these 3 clusters have little to zero behaviour once inside the honeypot (this is hinted by the fact that there were no downloaded files, no created files, and no executed commands), suggesting the attacks only used the honeypot to forward outgoing connections. Also, from the tree it is obvious that the main difference

between datapoints in cluster 5 and clusters 4 and 6 is the value of the variable *score_ip*. With the metrics shown in Table 1.4 we can confirm that cluster 5 is composed of datapoints that have the highest average of *score_ip*, compared to the other clusters, indicating the datapoints in this cluster have known untrustworthy IPs.

It seems cluster 6, the smallest cluster with just 1.52% of the data, contains datapoints for which the variable *duration_inside* is very large. In fact, we see in the decision tree there are two leaf nodes for cluster 6, both preceded by a rule comparing the variable *duration_inside*. There is another characteristic of cluster 6 the decision tree did not reflect: the variable *words_pass* (the average number of characters in a password) has a mean and median of 0, which means the logins to the system were made by leaving the password empty.

Cluster 7 shares some of the characteristics of cluster 2, but has two differences with respect to it: the amount of duration inside is considerably larger, with mean of 223.44 seconds (the mean of cluster 2 was 7.31). Also, the algorithm seems to have clustered the datapoints for which the number of commands executed *ncommands* is 55. In the tree we see the split rule $ncommands \leq 36$ separates cluster 7 from the rest of the nodes to its left.

Cluster 8 is characterised by datapoints with 0 outgoing connections, a slightly larger value of *duration_time_login* than the rest of the clusters, with an average 2.2215 seconds, and almost zero interaction with the honeypot once inside the honeypot (again, hinted by the fact that there were almost zero downloaded files, no created files, and a little number of executed commands).

		n_tries_logins	words_user	words_pass	duration_time_login	n_connections	duration_inside	nfiles_down	nfiles_create	ncommands	score_ip
Cluster 1	Min.	1.0000	4.0000	0.0000	0.2550	0.0000	3.0653	4.0000	0.0000	10.0000	0.0000
	Max.	1.0000	4.0000	10.0000	4.5341	1.0000	277.6225	8.0000	0.0000	55.0000	0.0000
	1st Qu.	1.0000	4.0000	5.0000	0.4234	0.0000	8.4126	5.0000	0.0000	14.0000	0.0000
	Median	1.0000	4.0000	5.0000	1.0071	0.0000	63.1611	6.0000	0.0000	16.0000	0.0000
	Mean	1.0000	4.0000	5.6502	1.0340	0.0076	81.5333	6.3498	0.0000	19.3536	0.0000
	3rd Qu.	1.0000	4.0000	7.0000	1.4723	0.0000	133.4819	8.0000	0.0000	26.0000	0.0000
Cluster 2	Min.	1.0000	4.0000	0.0000	0.1752	1.0000	0.6179	0.0000	0.0000	0.0000	0.0000
	Max.	1.0000	4.0000	11.0000	34.6631	2.0000	764.5872	0.0000	0.0000	10.0000	41.0000
	1st Qu.	1.0000	4.0000	0.0000	0.2792	1.0000	1.0670	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	5.0000	0.5714	1.0000	2.5914	0.0000	0.0000	0.0000	0.0000
	Mean	1.0000	4.0000	3.6744	0.7855	1.0058	7.3140	0.0000	0.0000	0.0105	1.8600
	3rd Qu.	1.0000	4.0000	5.0000	1.3272	1.0000	5.9589	0.0000	0.0000	0.0000	0.0000
Cluster 3	Min.	1.0000	4.0000	0.0000	0.1184	0.0000	0.6647	0.0000	0.0000	5.0000	0.0000
	Max.	1.0000	4.0000	12.0000	37.0029	0.0000	6521.4258	3.0000	1.0000	31.0000	100.0000
	1st Qu.	1.0000	4.0000	4.0000	0.2144	0.0000	11.2008	1.0000	0.0000	19.0000	0.0000
	Median	1.0000	4.0000	4.0000	0.2835	0.0000	31.1457	2.0000	0.0000	19.0000	0.0000
	Mean	1.0000	4.0000	3.9568	1.1139	0.0000	42.7999	1.3975	0.0042	18.4505	0.2176
	3rd Qu.	1.0000	4.0000	5.0000	0.6118	0.0000	31.9678	2.0000	0.0000	19.0000	0.0000
Cluster 4	Min.	1.0000	4.0000	0.0000	0.2263	3.0000	0.9421	0.0000	0.0000	0.0000	0.0000
	Max.	1.0000	4.0000	5.0000	30.6465	5.0000	5419.0046	1.0000	0.0000	17.0000	5.0000
	1st Qu.	1.0000	4.0000	0.0000	0.3285	3.0000	1.7574	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	5.0000	0.6206	3.0000	3.0000	0.0000	0.0000	0.0000	0.0000
	Mean	1.0000	4.0000	2.6196	1.1228	3.1603	32.6508	0.1459	0.0000	2.4809	0.0120
	3rd Qu.	1.0000	4.0000	5.0000	0.9221	3.0000	12.6928	0.0000	0.0000	0.0000	0.0000
Cluster 5	Min.	1.0000	4.0000	0.0000	0.2627	0.0000	1.1300	0.0000	0.0000	0.0000	41.0000
	Max.	2.0000	4.0000	7.0000	3.7088	4.0000	1009.1308	0.0000	0.0000	0.0000	100.0000
	1st Qu.	1.0000	4.0000	0.0000	0.3063	4.0000	1.3426	0.0000	0.0000	0.0000	41.0000
	Median	1.0000	4.0000	4.5000	0.3228	4.0000	2.2717	0.0000	0.0000	0.0000	41.0000
	Mean	1.0064	4.0000	2.5892	0.5332	3.8025	18.9342	0.0000	0.0000	0.0000	43.2229
	3rd Qu.	1.0000	4.0000	5.0000	0.4003	4.0000	7.2942	0.0000	0.0000	0.0000	41.0000
Cluster 6	Min.	1.0000	4.0000	0.0000	0.1695	0.0000	10761.8109	0.0000	0.0000	0.0000	0.0000
	Max.	1.0000	4.0000	0.0000	2.8687	5.0000	14824.1145	0.0000	0.0000	0.0000	0.0000
	1st Qu.	1.0000	4.0000	0.0000	0.7154	0.0000	11675.8240	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	0.0000	1.1028	3.0000	11965.9135	0.0000	0.0000	0.0000	0.0000
	Mean	1.0000	4.0000	0.0000	1.0599	2.4706	12061.9427	0.0000	0.0000	0.0000	0.0000
	3rd Qu.	1.0000	4.0000	0.0000	1.2061	4.0000	12200.0996	0.0000	0.0000	0.0000	0.0000
Cluster 7	Min.	1.0000	4.0000	3.0000	0.1955	1.0000	182.3130	0.0000	0.0000	55.0000	0.0000
	Max.	1.0000	4.0000	11.0000	45.5291	1.0000	802.2293	0.0000	0.0000	55.0000	0.0000
	1st Qu.	1.0000	4.0000	5.0000	0.3305	1.0000	190.5036	0.0000	0.0000	55.0000	0.0000
	Median	1.0000	4.0000	6.0000	0.3369	1.0000	205.2743	0.0000	0.0000	55.0000	0.0000
	Mean	1.0000	4.0000	6.6018	0.5893	1.0000	224.4392	0.0000	0.0000	55.0000	0.0000
	3rd Qu.	1.0000	4.0000	8.0000	0.3493	1.0000	231.3169	0.0000	0.0000	55.0000	0.0000
Cluster 8	Min.	1.0000	4.0000	0.0000	0.0844	0.0000	0.5859	0.0000	0.0000	0.0000	0.0000
	Max.	7.0000	7.0000	56.0000	84.5535	0.0000	6987.3672	1.0000	0.0000	12.0000	79.0000
	1st Qu.	1.0000	4.0000	4.0000	0.5183	0.0000	1.7148	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	5.0000	1.3150	0.0000	3.9360	0.0000	0.0000	0.0000	0.0000
	Mean	1.0526	4.0042	5.2749	2.2215	0.0000	29.6373	0.0145	0.0000	0.3043	0.1655
	3rd Qu.	1.0000	4.0000	7.5000	2.0680	0.0000	15.1657	0.0000	0.0000	0.0000	0.0000

Table 1.4: Summary metrics calculated for all datapoints of each of the clusters obtained with the k-means algorithm.

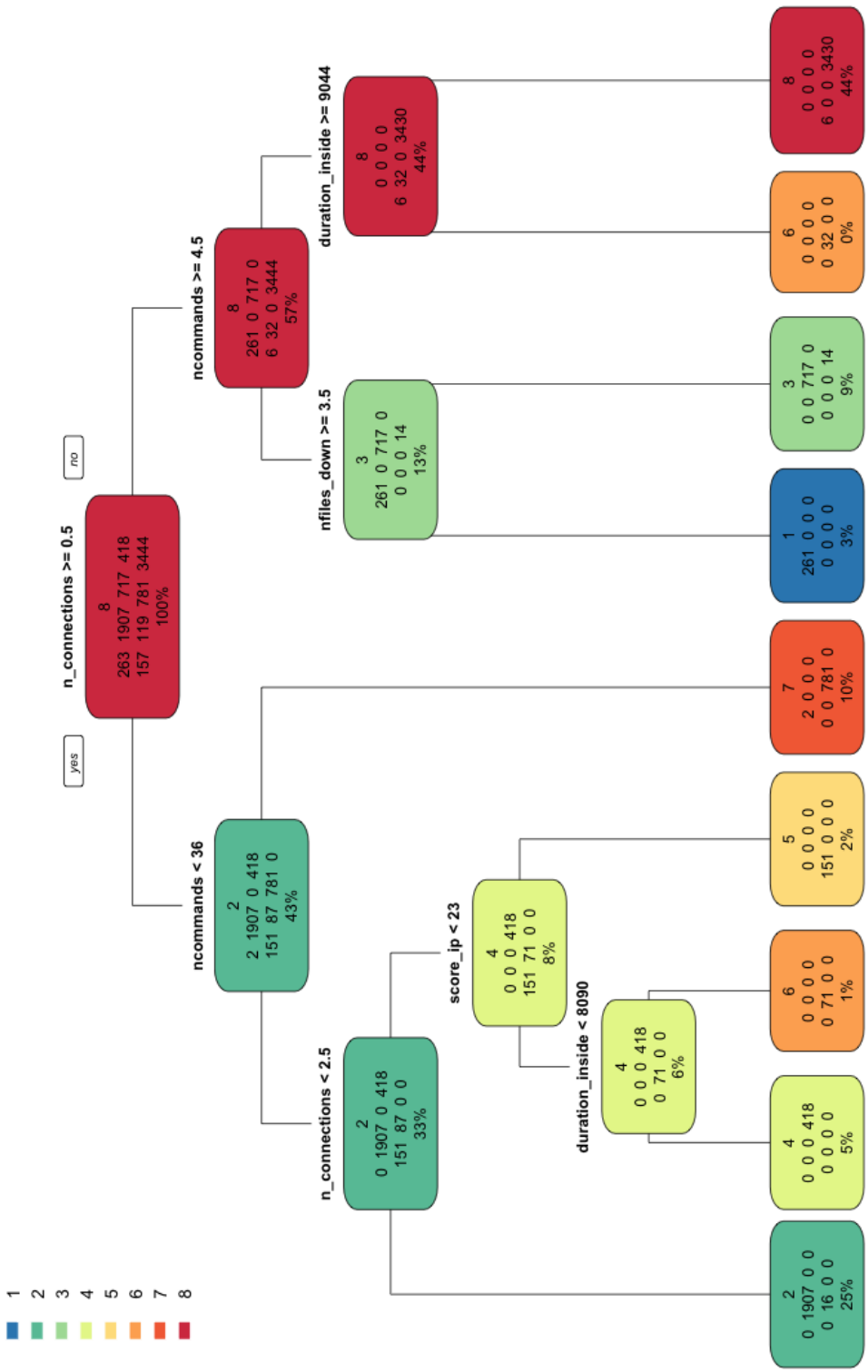


Figure 1.10: Decision tree trained using the clusters found by the k-means algorithm as the data labels.

1.2.3 K-prototypes clustering results

In this section we trained a k-prototypes algorithm, as described in Section 1.1.2, to find clusters in the data collected by a honeypot. The main idea behind this algorithm, is to be able to include the *sshversion* categorical variable in the analysis. This variable is important because it can be used to identify the operating system from which the attacks were originated, since different operating systems make use of different versions of the SH protocol. This extra information may be used by a cybersecurity expert to better understand the nature of the attacks.

Given the limitation of the k-means algorithm to handle categorical variables, we were motivated to use another clustering algorithm that was able to include categorical variables in its analysis. We used the k-prototypes algorithm to analyse the honeypot data using all the variables listed in Table 1.1. For the analysis in this section, we used several functions of the *clustMixType* package in R.

Data preprocessing

The k-prototypes algorithm, similarly to the k-means algorithm, is also susceptible to the different scales of the numerical variables, as it also uses the squared Euclidean distance in its calculations. The same approach used to solve this issue in k-means was also used in this section, that is, we transformed the numerical variables by dividing them by its range.

Given that k-prototypes is able to handle categorical variables, we were able to include the variable *sshversion* as a factor in the computation of the algorithm.

Tuning the k-prototypes algorithm

For the k-prototypes algorithm, we also need to specify the value of the hyperparameter k, the number of clusters to be formed by the algorithm. Additionally, the k-prototypes algorithm also needs a value for γ, the variable which controls the weight of the categorical variables in the algorithm, as given by Equation 1.1. The function *lambdaest* (from the *clustMixType* package) was used for the estimation of the value of γ. The estimated value by the *lambdaest* function is $\gamma = 0.022902$. This value, being less than one, suggests very little importance is given to the variable *sshversion*, the only categorical feature in the data.

In order to find the number of clusters, k, we used the same K-fold cross validation approach used for tuning the k-means algorithm in Section 1.2.2: we tuned the k-prototypes algorithm by perfoming 10-fold cross validation on the dataset. On each fold, a k-prototypes model was trained using the corresponding training set, and then used the test set to evaluate the model performance using the Silhouette score introduced in Section 1.1.1. Finally, we calculated the average of the 10 Silhouettes scores found via cross validation, to obtain the final estimation of the Silhouette score. We did this for values of $k \in \{2 \dots, 10\}$. We capped the number of clusters at 10, because we found that increasing the number of clusters beyond 10 did not improve the Silhouette score of the clusters that were formed.

The results of the 10-fold cross validation used for tuning the hyperparameter k can be seen in Table 1.5. A plot of the values is shown in Figure 1.11. The results show that the best value for the number of clusters in k-prototypes, according to the Silhouette score, is $k = 2$, with a Silhouette mean score of 0.7976. The second best result is $k = 4$, with a Silhouette mean score of 0.7688. We see that

Num. of centers (k)	Mean test Silhouette score
2	0.7976*
3	0.7564
4	0.7688
5	0.7123
6	0.5850
7	0.6403
8	0.5714
9	0.5857
10	0.5093

Table 1.5: Silhouette test means obtained from applying 10-fold cross validation to the data. The best model according to the scores is $k = 2$, followed by $k = 4$

for larger values of the number of clusters, the value of the Silhouette metric keeps decreasing.

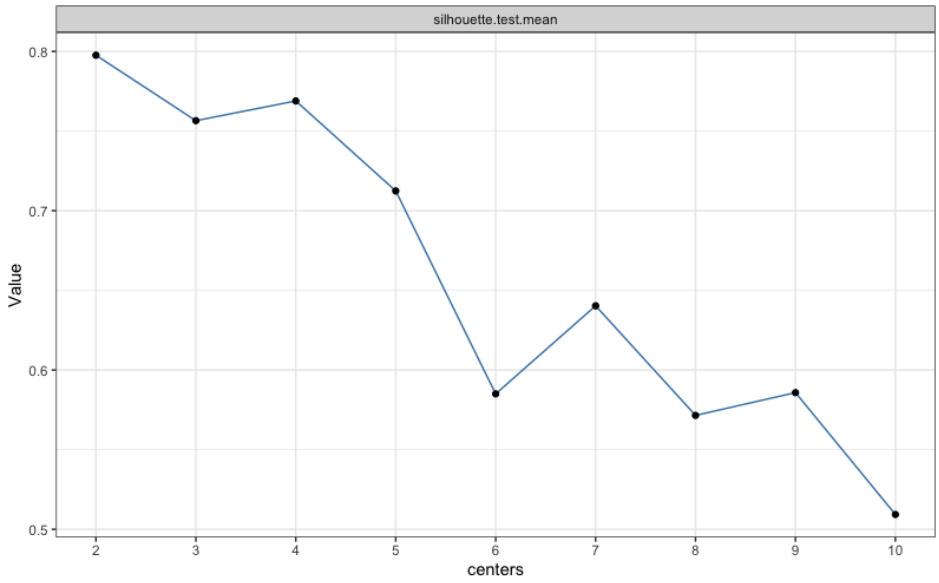


Figure 1.11: Plot of the mean test Silhouette scores calculated from the 10-fold cross validation approach, used for tuning the number of clusters in the k-prototypes algorithm.

Training the k-prototypes model

We continued the analysis by training a k-prototypes model with $k = 2$, the best value for the number of centers found by the cross validation approach. The algorithm partitioned the data into 2 clusters. Table 1.6 shows the number of datapoints assigned to each cluster. We see that cluster 2 is significantly larger than cluster 1, containing 89.92% of the datapoints.

Cluster number	1	2
Number of datapoints	787	7019
Percentage of data	10.08%	89.92%

Table 1.6: Number of datapoints in each cluster. The last row shows the percentage of all the data contained in each cluster.

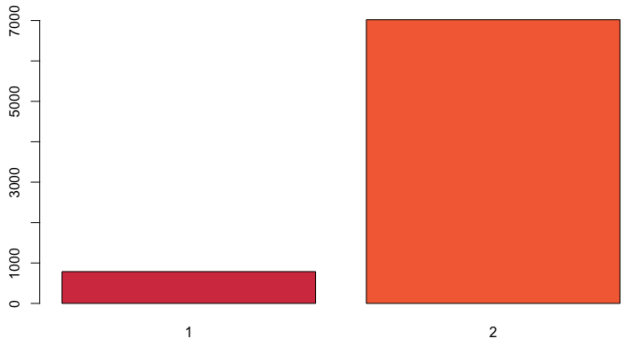


Figure 1.12: Table plot of the number of datapoints assigned to each cluster, for $k = 2$.

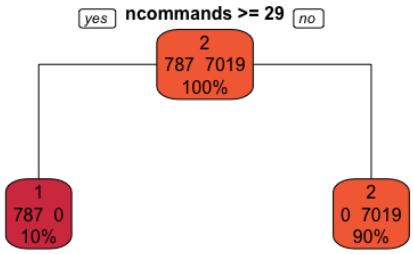


Figure 1.13: This is the decision tree model obtained by using the clusters formed by the k-prototypes algorithm, with $k = 2$, as the labels (target variables) for the data.

A decision tree was trained using the clusters obtained from the k-prototypes algorithm as the target variables. As mentioned earlier, decision trees provide for a visual aid to interpret the results of the clusters formed by the k-prototypes algorithm. The decision tree obtained from the data is shown in Figure 1.13. It shows the data can be partitioned with a single rule: datapoints are assigned to either cluster depending on the number of commands that were executed, given by the variable *ncommands*. If $ncommands \geq 29$ the datapoint is assigned to cluster 1, otherwise it is assigned to cluster 2.

This particular partition of the data into two clusters does not provide enough useful information about the data. Even though we learned the variable *ncommands* is important, it is not sufficient to develop useful insights about the structure of the data. Because of this, we decided to train a second k-prototypes model using the second best option found for the number of clusters, which was $k = 4$.

Cluster number	1	2	3	4
Number of datapoints	657	829	781	5539
Percentage of data	8.42%	10.62%	10.00%	70.96%

Table 1.7: Number of datapoints in each cluster after training a k-prototypes model with $k = 4$.

Figure 1.14 shows the number of datapoints in each cluster resulting form training the model again with $k = 4$. We see the clusters are still imbalanced in terms of the number of elements assigned to them. Cluster 4 contains 5539 datapoints with makes up for 70.96% of the total datapoints. The smallest cluster is 1 with just 8% of the data; clusters 2 and 3 have approximately the same size, containing around 10% of the datapoints. A summary of some statistics calculated with the datapoints contained in each cluster are shown in Table 1.8.

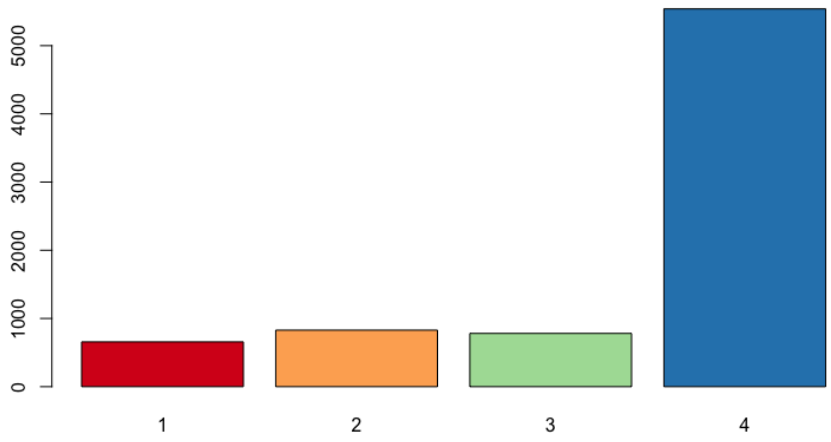


Figure 1.14: Table plot of the clusters found in the data with $k = 4$.

A second decision tree was trained using the 4 clusters assignments found by the k-prototypes algorithm as the target variables. The trained decision tree model is depicted in Figure 1.15. By looking at the decision tree we can see several interesting things. Out of the 11 variables used to train the clustering algorithm, the decision tree found that only 4 were the most relevant: *ncommands*, *n_connections*, *nfiles_down*, and *sshversion*. The *ncommands* variable appears three times in the decision tree, and *n_connections* appears twice. We can see the first rule found by the decision tree splits the root node by the number of commands executed, the *ncommands* variable, the same as the rule found by the tree trained for the previous model with $k = 2$. However, the threshold value has changed from 29 to 6.5. Also, 6 out of 8 of the leaf nodes in the decision tree are all pure, meaning the decision tree was able to find rules that partition the datapoints according to their labels given by the clusters. Additionally, the decision tree made a split using the *sshversion* variable. This split is not very useful if we use it to differentiate between datapoints belonging to cluster 2 and cluster 4: it is only able to tell apart 6 datapoints of cluster 4 from cluster 2. However, this split does give us some information about the datapoints assigned to cluster 2: at least 802 datapoints of cluster 2 belong to one of the two categories of the *sshversion* variable, *kthrss__x00* or *libssh*.

From the rules in the decision tree and the information shown on Table 1.8, we see cluster 1 is characterised by having the datapoints for which the number of outgoing connections, *n_connections*, are 2 or more, having a mean of 3.3851, the highest among all the clusters. This cluster gathered the datapoints that represent the attacks which used the honeypot to forward multiple outgoing connections.

Cluster 2 contains datapoints which had a large number of commands executed, *ncommands*, having an average of 18.99. It is also the cluster that had the most number of file downloads (*nfiles_down*), with an average of 3.2075, a lot more than the other clusters. This cluster is very similar to cluster 1 found by the k-means algorithm which also had a large number of commands executed and downloaded files.

Cluster 3 contains the datapoints for which the number of executed commands, *ncommands*, is exactly 55 commands, the number of outgoing connections, *n_connections* is 1, and, looking at the median of *duration_inside*, spent the most time inside the session compared to the other clusters. We see cluster 3 matches cluster 7 from the results of the k-means algorithm.

Cluster 4, the largest cluster containing 70.96% of the datapoints, is characterised by having a small amount of connections (1 or less), and little to no behaviour inside the session (given the small values for the mean and median of *nfiles_down*, *nfiles_create*, and *ncommands*).

Overall, it seems the results of the k-prototypes clustering algorithm did not improve the results obtained through the k-means algorithm. The k-means algorithm produced more clusters that were able to gather datapoints with specific characteristics. The inclusion of the categorical variable *sshversion* by using the k-prototypes algorithm did not bring any useful information.

cluster_text	rows	n_tries_logins	words_user	words_pass	duration_time_login	n_connections	duration_inside	nfiles_down	nfiles_create	ncommands	score_ip
Cluster 1	Min.	1.0000	4.0000	0.0000	0.1695	2.0000	0.8146	0.0000	0.0000	0.0000	0.0000
	Max.	1.0000	4.0000	11.0000	30.6465	5.0000	14824.1145	1.0000	0.0000	17.0000	41.0000
	1st Qu.	1.0000	4.0000	0.0000	0.3163	3.0000	1.7471	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	0.0000	0.5947	3.0000	3.6785	0.0000	0.0000	0.0000	0.0000
	Mean	1.0000	4.0000	2.3425	0.9605	3.3851	1429.2102	0.0928	0.0000	1.5784	9.4429
	3rd Qu.	1.0000	4.0000	5.0000	0.9241	4.0000	28.5208	0.0000	0.0000	0.0000	0.0000
	Min.	1.0000	4.0000	0.0000	0.1184	0.0000	0.8519	0.0000	0.0000	7.0000	0.0000
	Max.	1.0000	4.0000	11.0000	37.0029	1.0000	5618.7692	8.0000	1.0000	55.0000	100.0000
Cluster 2	1st Qu.	1.0000	4.0000	4.0000	0.2245	0.0000	11.6248	2.0000	0.0000	18.0000	0.0000
	Median	1.0000	4.0000	4.0000	0.3857	0.0000	31.2846	2.0000	0.0000	19.0000	0.0000
	Mean	1.0000	4.0000	4.6815	1.1475	0.0024	51.6249	3.2075	0.0036	18.9891	0.1834
	3rd Qu.	1.0000	4.0000	6.0000	1.0112	0.0000	41.8348	5.0000	0.0000	19.0000	0.0000
	Min.	1.0000	4.0000	3.0000	0.1955	1.0000	182.3130	0.0000	0.0000	55.0000	0.0000
	Max.	1.0000	4.0000	11.0000	45.5291	1.0000	802.2293	0.0000	0.0000	55.0000	0.0000
Cluster 3	1st Qu.	1.0000	4.0000	5.0000	0.3305	1.0000	190.5036	0.0000	0.0000	55.0000	0.0000
	Median	1.0000	4.0000	6.0000	0.3369	1.0000	205.2743	0.0000	0.0000	55.0000	0.0000
	Mean	1.0000	4.0000	6.6018	0.5893	1.0000	224.4392	0.0000	0.0000	55.0000	0.0000
	3rd Qu.	1.0000	4.0000	8.0000	0.3493	1.0000	231.3169	0.0000	0.0000	55.0000	0.0000
	Min.	1.0000	4.0000	0.0000	0.0844	0.0000	0.5859	0.0000	0.0000	0.0000	0.0000
	Max.	7.0000	7.0000	56.0000	84.5535	1.0000	14171.0184	2.0000	0.0000	20.0000	100.0000
Cluster 4	1st Qu.	1.0000	4.0000	2.0000	0.3435	0.0000	1.5393	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	5.0000	0.8929	0.0000	3.1281	0.0000	0.0000	0.0000	0.0000
	Mean	1.0329	4.0026	4.6178	1.6819	0.3441	115.2470	0.0114	0.0000	0.6581	0.8500
	3rd Qu.	1.0000	4.0000	6.0000	1.6386	1.0000	13.5636	0.0000	0.0000	0.0000	0.0000

Table 1.8: Summary metrics calculated for all points of each of the clusters obtained with the k-prototypes algorithm.

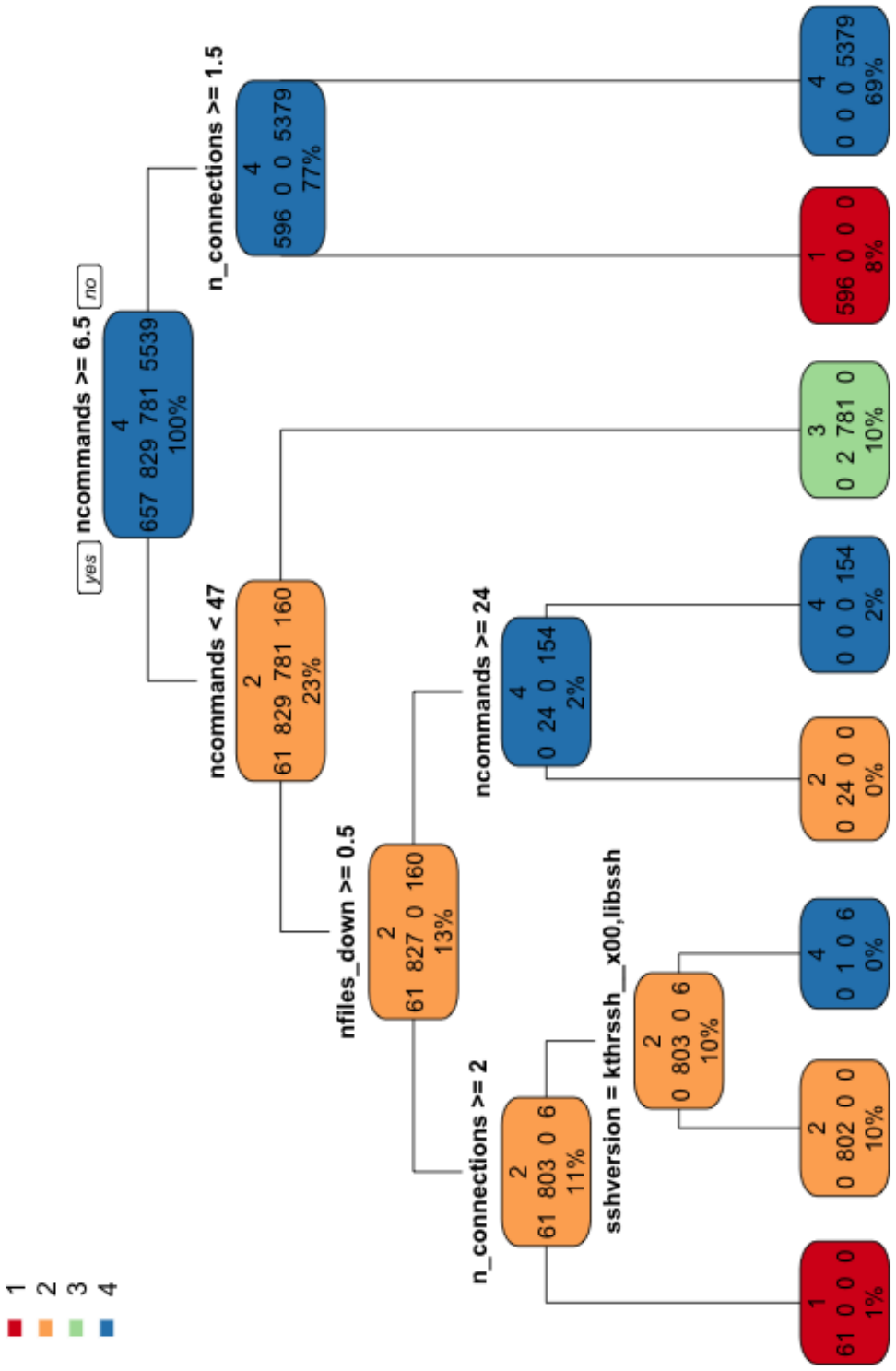


Figure 1.15: Resulting decision tree that was calculated using the clusters from using k-prototypes, with $k = 4$, as the labels.

1.2.4 UMAP dimensionality reduction with DBSCAN clustering results

In this section we used the dimensionality reduction algorithm, UMAP, as described in Section 1.1.3 in order to find the representation of data in two dimensions. Finally, we used the DBSCAN algorithm introduced in Section 1.1.3 on the resulting 2-dimensional embedding to find the clusters in the data. The motivation behind this is to see if we could improve the analysis by applying a dimensionality reduction algorithm prior to the clustering algorithm. We decided to use the DBSCAN algorithm for this task, because we thought it would perform well given the shapes obtained by reducing the dimensions of the data with the UMAP algorithm.

The analyses in this section were obtained using several functions in the *dbscan* and *umap* packages in R.

Data preprocessing

As mentioned earlier, the variable *sshversion* is of great interest in the cybersecurity context. Therefore, we would like to include it in the analysis. One way of achieving this is by using an encoding method, such as *dummy encoding*. Dummy encoding works by creating $c-1$ binary feature columns, where c is the number of levels of the categorical variable. Each of the new binary columns represents a possible category. A simple example of such encoding can be seen in Table 1.9, where a categorical value with three colour levels, blue, yellow, and red, is encoded into $c-1=3-1=2$ binary columns. The variable *sshversion* has 31 levels, meaning that $31-1=30$ binary features were added to the data.

Level	e_1	e_2
blue	1	0
yellow	0	1
red	0	0

Table 1.9: Dummy encoding example of a categorical variable with three colours (levels): blue, yellow, red. Columns e_1 and e_2 represent the encoded binary columns.

UMAP Dimensionality Reduction of the Data

Our next step in the analysis is to apply the UMAP algorithm to the data in order to reduce its number of dimensions to 2. To use the algorithm, we need to choose values for the *min_dist* and *n_neighbours* hyperparameters. The way we approached this is by creating a grid with some possible combinations of both variables, and ran the UMAP algorithm with the data for every combination. Figure 1.16 shows the effects of different combinations of the values for both hyperparameters; on the columns we see the effect of increasing *min_dist* from 0.1 to 1 in leaps of 0.3, and on the rows we see the effect of increasing the value of *n_neighbours* starting at 10 and finishing at 100, in increments of 30. As the value of *n_neighbours* increases, we can se the algorithm is connecting more neighbouring points together, which improves the structure of the projection of the data. The *min_dist* parameter controls how "spread" the points are from one another as it represents the minimum distance apart that points are allowed to be in the low dimensional representation; we see that for larger values, the values are more spread out.

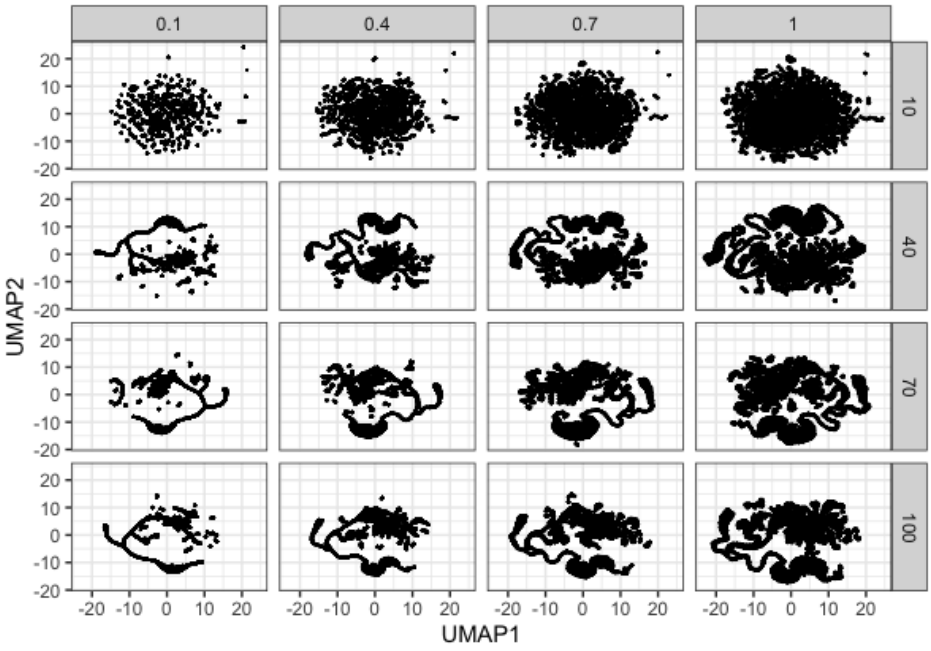


Figure 1.16: Graph showing the different results obtained from the combinations of the UMAP hyperparameters *n_neighbours* (row facets), and *min_dist* (column facets).

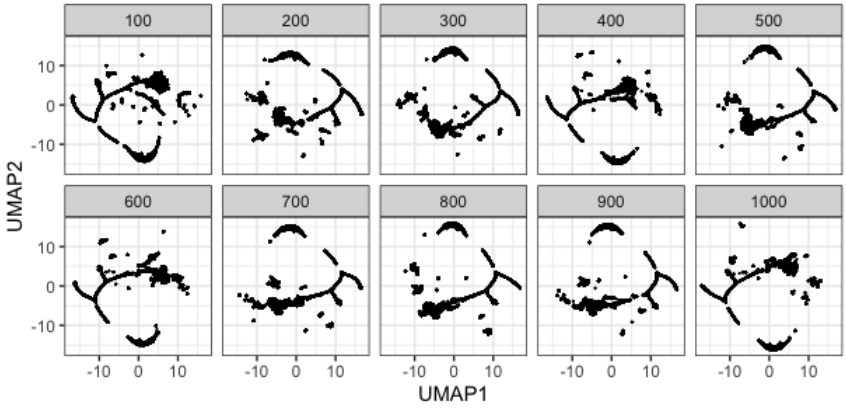


Figure 1.17: Comparison of the results obtained for different values of *n_neighbours*, when the value of *min_dist* is fixed to 0.001

Given the results it is clear we should prefer larger values for *n_neighbours* and smaller values for *min_dist*. According to McInnes et al. (2018), in clustering applications it is beneficial to set *min_dist* to a very low value, as that will make points to remain packed together increasing the density of the projections, and it will also make cleaner separations between clusters. We ran the UMAP algorithm for values of $n_neighbours \geq 100$, and fixed the value of *min_dist* to 0.001. The results can be seen in 1.17. Through visual inspection, it is not very clear which value of *n_neighbours* to select. We selected the UMAP projection with $n_neighbours = 400$, and $min_dist = 0.001$ to continue with the analysis, as it seems these parameter values provide for a good representation of the data. We can see

the plot of the selected UMAP results in Figure 1.18.

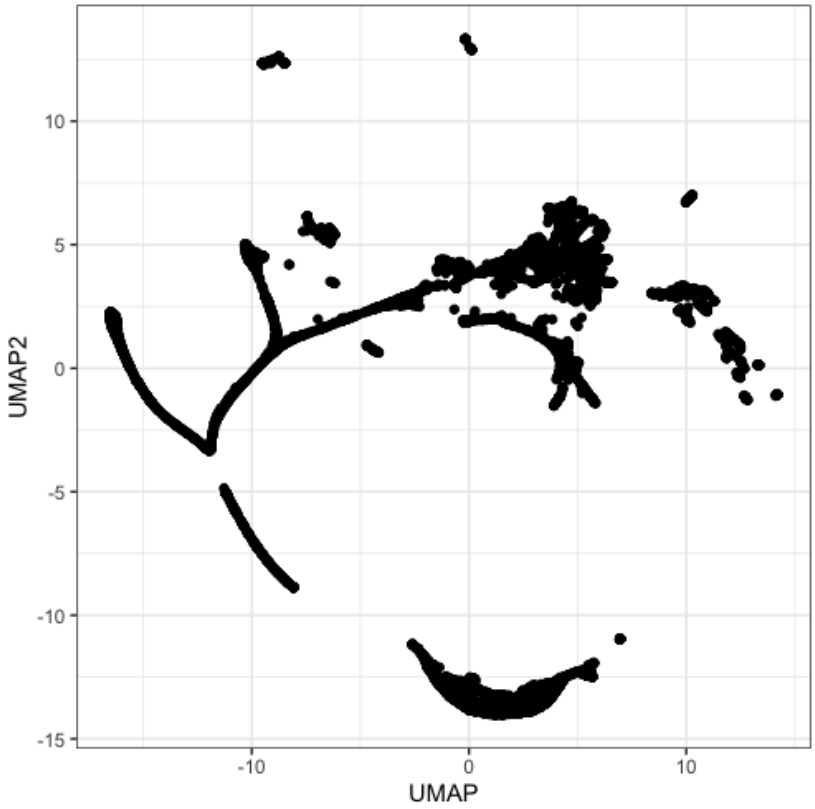


Figure 1.18: Graph of the results of UMAP algorithm with $n_neighbours = 400$ and $min_dist = 0.001$.

Selection of DBSCAN Parameters

Once we obtained the 2-dimensional representation of the data by using the UMAP algorithm, we continued the clustering analysis with the DBSCAN algorithm. First, we needed to select appropriate values for hyperparameters $minPts$ and eps. The value for $minPts$ is easy to select if we follow the heuristic introduced in Section 1.1.3. Because we have reduced the dimensionality of the data to 2 dimensions, we can choose the value as $minPts = 4$.

We are left with the selection of eps. To choose a value for the parameter we need to compute the k-distance graph, which is the sorted distance from each point to its kth-nearest neighbour. The *kNNdistplot* function from the *dbscan* package makes this computations for us. The resulting k-distance graph is shown in Figure 1.19. In the attempt of automatising as much as the analysis as possible, we made use of the python implementation Arvai (2020) of the *kneedle* algorithm (Satopaa et al., 2011), which approximates the knee points for any given dataset. The algorithm has a sensitivity parameter, S, which adjusts how aggressive the algorithm is when detecting knees. Smaller values of S detect knees quicker, while larger values are more conservative. The value of the parameter was set to its default, $S = 1$, which in consequence detects the beginning of the curvature. On the graph, we can see that the algorithm detected the beginning of the curve, when looked from right to left, at the value of

0.2963. Therefore, we set the value of $eps = 0.2962$.

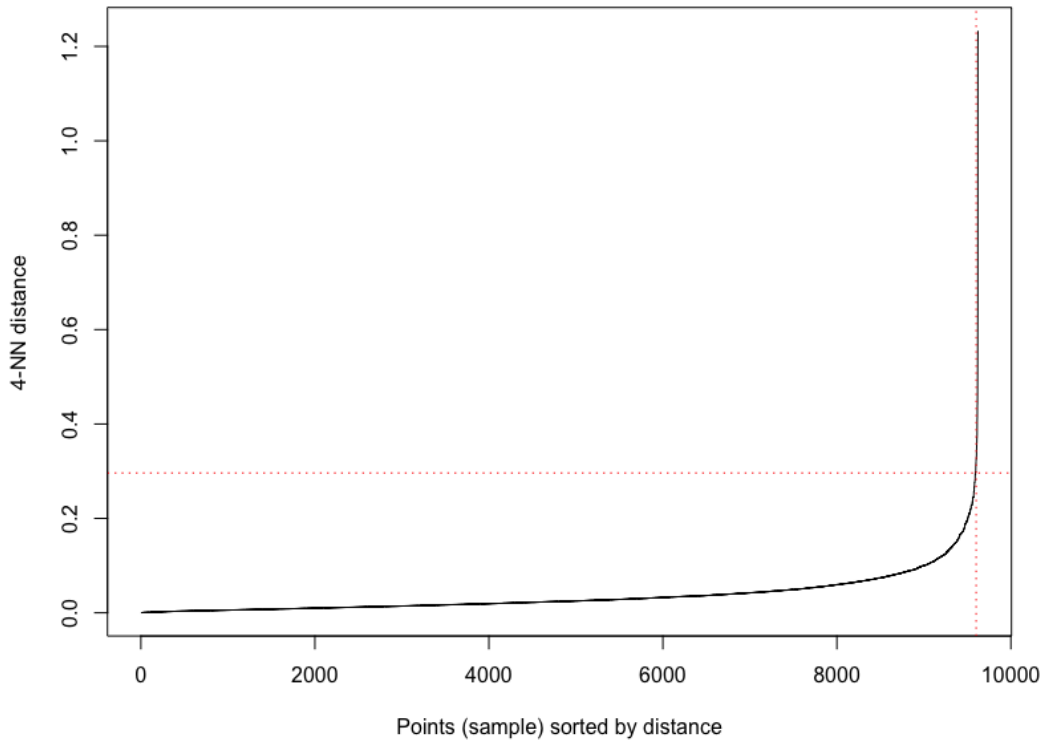


Figure 1.19: k-distance graph used to select the value of the parameter eps. The intersection of the dotted lines shows the for which the $Kneedle$ algorithm detects the $knee$ in the plot at a value of approximately 0.2963.

We used the *dbscan* function from the *dbscan* package in R, to make the calculations of the clustering algorithm, with the hyperparameters $minPts = 4$ and $eps = 0.2963$. The DBSCAN model found 24 different clusters of data, plus one extra cluster composed of outlier datapoints. The clustering results applied to the 2-dimensional UMAP projection of the data can be seen in Figure 1.20. The algorithm was able to detect 13 datapoints as outliers, which were assigned to cluster 0, and are depicted in grey in the graph. These points were not reachable by any other point, or did not have the minimum number of points, specified by $minPts$, in their eps-neighbourhood. Consequently, the algorithm could not assign them to any cluster. Figure 1.21 shows the distribution of the datapoints in each of the clusters.

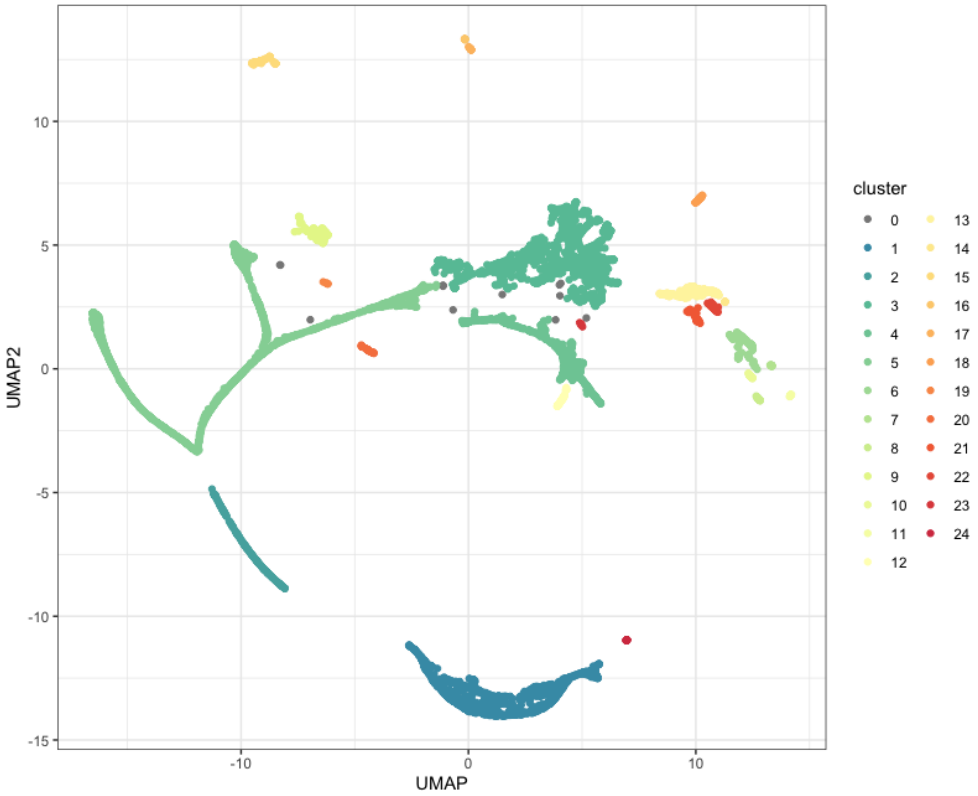


Figure 1.20: Graph showing the results from applying the DBSCAN clustering algorithm, with parameters $minPts = 4$ and $eps = 0.2963$, to the UMAP projection of the data. The points that were classified as outliers are assigned to cluster 0, and are shown in grey.

We can observe that the majority of the datapoints were assigned to a few big clusters, while the remaining datapoints were distributed in very small clusters. Table 1.10 shows how many datapoints were assigned to each cluster, as well as the respective percentage of the data. We see that cluster number 3 is the largest containing 26.76% of the datapoints, followed by cluster 5 with 21.94%, and cluster 1 with 16.72%. These three clusters together contain more than 60% of the data. On the other hand, there are many clusters than each have less than 1% of the datapoints. Clusters 13 and 19, for example, have 0.14% and 0.08% of the data, respectively. Cluster 23 is the smallest one with just 5 points (0.05%).

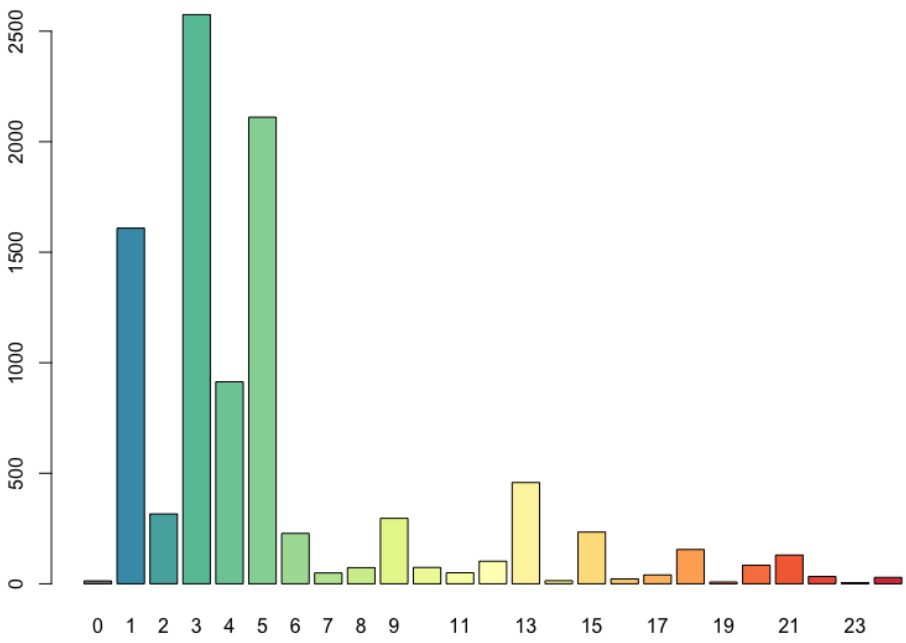


Figure 1.21: Table plot of the clusters found by the DBSCAN algorithm, with parameters $minPts = 4$ and $eps = 0.2963$

Cluster number	Number of datapoints	Percentage of data
0	13	0.14 %
1	1609	16.72 %
2	316	3.28 %
3	2575	26.76 %
4	914	9.50 %
5	2111	21.94 %
6	228	2.37 %
7	49	0.51%
8	73	0.76%
9	296	3.08%
10	74	0.77%
11	50	0.52%
12	102	1.06%
13	458	4.76 %
14	14	0.15%
15	234	2.43 %
16	22	0.23 %

Cluster number	Number of datapoints	Percentage of data
17	40	0.42 %
18	155	1.61 %
19	8	0.08 %
20	84	0.87 %
21	130	1.35%
22	33	0.34%
23	5	0.05%
24	29	0.3%

Table 1.10: Number of datapoints and their corresponding percentage in each cluster. Cluster 0 contains the points that were detected as outliers by the algorithm.

The strategy we followed to proceed with the analysis was to keep the n largest clusters that contain more than 90% of the data when accumulated, and group the remaining clusters into a single cluster. In order to accomplish this, the clusters we ordered from biggest to smallest in terms of the datapoints they contained. Then we calculated the cumulative proportion of data, as shown in Figure 1.22. The graph shows that the first 9 clusters contain more than 90% of the data. After these 9 clusters, the cumulative proportion of data continues to grow higher but in very small increments, until it reaches 1 (100%) when all the datapoints of the clusters are considered.

Figure 1.23 shows the new distribution of the datapoints in the 9 largest clusters that we kept from the original DBSCAN clustering, plus cluster 0, which now contains the rest of the datapoints. Table 1.11 shows the number of datapoints assigned to each cluster, as well as their percentage of the total data. The smallest cluster now contains 228 datapoints (2.37% of the data) and cluster 0 contains 881 datapoints, corresponding to less than 10% of the data.

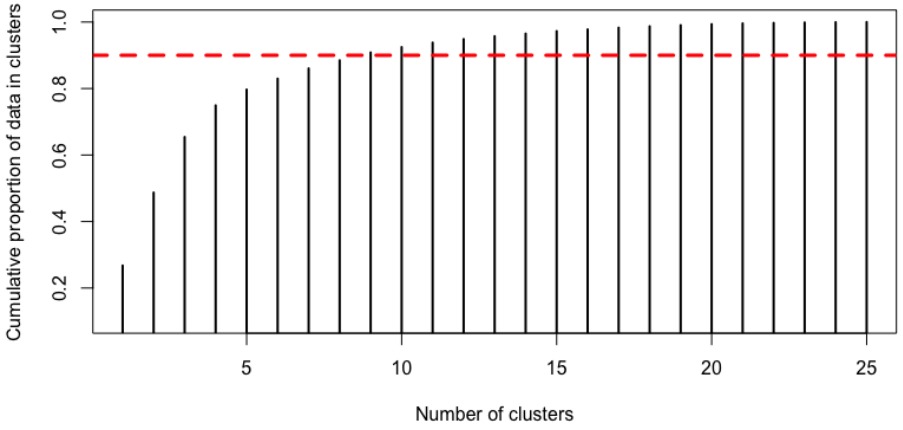


Figure 1.22: Graph of the cumulative proportion of data contained in the clusters. The red dotted line is at 0.9, which denotes the threshold of the cumulative 90% of the data. We can see that the threshold is crossed by the $9th$ accumulated cluster.

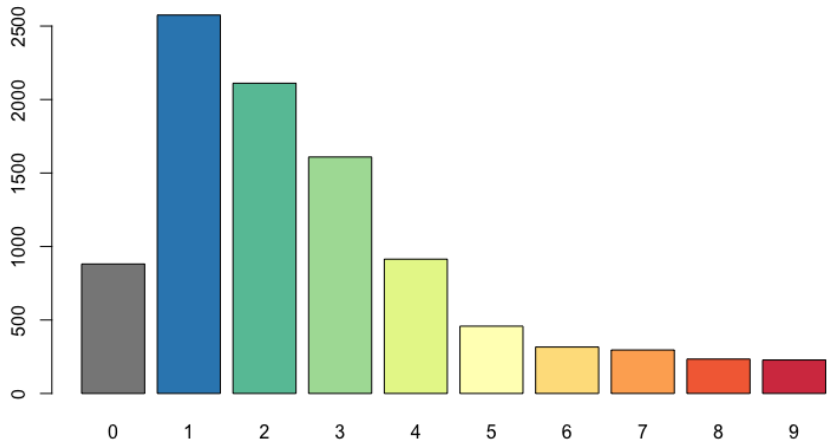


Figure 1.23: Table plot of the clusters found by the DBSCAN algorithm, after keeping the 9 biggest clusters, and grouping the rest into the outlier cluster (cluster 0).

The resulting cluster assignation can be seen in Figure 1.24, where the 9 biggest clusters remain unchanged, while all the others got reassigned to cluster 0. Similarly to what was done in the previous sections, we trained a decision tree based on the clusters obtained, shown in Figure 1.25. Table 1.12 shows the summary statistics computed for each of the 9 clusters.

Cluster number	Number of datapoints	Percentage of data
0	881	9.16 %
1	2575	26.76 %
2	2111	21.94 %
3	1609	16.72 %
4	914	9.50 %
5	458	4.76 %
6	316	3.28 %
7	296	3.08 %
8	234	2.43 %
9	228	2.37 %

Table 1.11: Number of datapoints and their corresponding percentage in each cluster, after reassigning all the smallest clusters into the cluster 0, and keeping the 9 largest clusters.

In Figure 1.24 we can see several well-separated groups, with some interesting shapes. Cluster 1, shown in blue in Figure 1.24, is the largest cluster containing 26% of the datapoints. It is characterised by having larger values of the *duration_time_login* than the rest of clusters, with a median of 1.4649 seconds. If we look at the splitting rules learned by the decision tree, we see that the largest leaf note corresponding to cluster 1 contains datapoints with an average of password lengths equal or greater than 2, most of the datapoints in the cluster do not have the *sshversion* value of *Granados*.1.0, executed less than 10 commands (*ncommands*), and their score ip was less than 29. Actually, if we look at the table the median value of both variables *ncommands* and *score_ip* is 0.

Cluster 2, shown in a dark shade of green in Figure 1.24. This cluster has higher values of *duration_inside* when compared to the rest of clusters, although, it still has smaller values than those of clusters 3 and 6. This fact is supported by the decision tree (Figure 1.25), where the leaf nodes corresponding to clusters 2, 3, and 6, fall on the same side (to the left) of the rule at the root node $duration_inside \geq 22$. Another characteristic is that it gathered datapoints which executed a high number of commands (*ncommands*), with a mean of 25.8470 commands. From Table 1.12, we see that cluster 7 also had a high number of executed commands, however, the values of duration inside are not as large as those of number cluster 2.

Cluster 3, shown at the bottom of Figure 1.24 in light green, is clearly separated from the rest of the points. Looking at the summary statistics on Table 1.12, we can see that it is characterised by having the largest times of *duration_inside* among all clusters. The median of the variable *words_pass*, the average length of passwords used in the login, is 0, suggesting that most of the datapoints in this cluster logged in to the honeypot by using a combination of a username with an empty password. The datapoints in this cluster also have a large number of outgoing connections ($n_connections$), with a median of 58.0000. and had no activity inside the session, indicated by the 0 values in the mean and median statistics of the variables $nfiles_down$, $nfiles_create$ and *ncommands*. It is not unusual for datapoints with a high number of connections to have high values of the variable *duration_inside*, which is the case for cluster 3. This indicated that attacks gathered in this cluster had the goal of creating a large number of outgoing connections, which increased the amount of time spent inside the honeypot. The data in this cluster closely resembles that of cluster 6 of the k-means algorithm, as both clusters share the characteristics of having high values of the *duration_inside* variable and used empty passwords when logging in.

Cluster 4 had, on average, small values for the *words_pass* variable, implying the attacks used empty passwords to log into the honeypot. It also had small values of the variables *duration_time_login*, and *duration_inside*. The datapoints also show little to no interaction by having small values of number of files downloaded ($nfiles_down$), number of created files ($nfiles_create$), and number of executed commands($ncommands$). These characteristics indicate that the attacks were probably made by a script, which logged in using the some username with no password, and after a few seconds, logged out of the system.

From the decision tree in Figure 1.25 we can see that one of the main characteristics of clusters 5 and 9 is that its datapoints share the same *sshversion*, *Granados*.1.0. Also both clusters contain datapoints which performed a single outgoing connection ($n_connections$). However, the main difference between these two clusters is that on average, the amount of time of *duration_inside* is slightly larger for cluster 5 than for cluster 9.

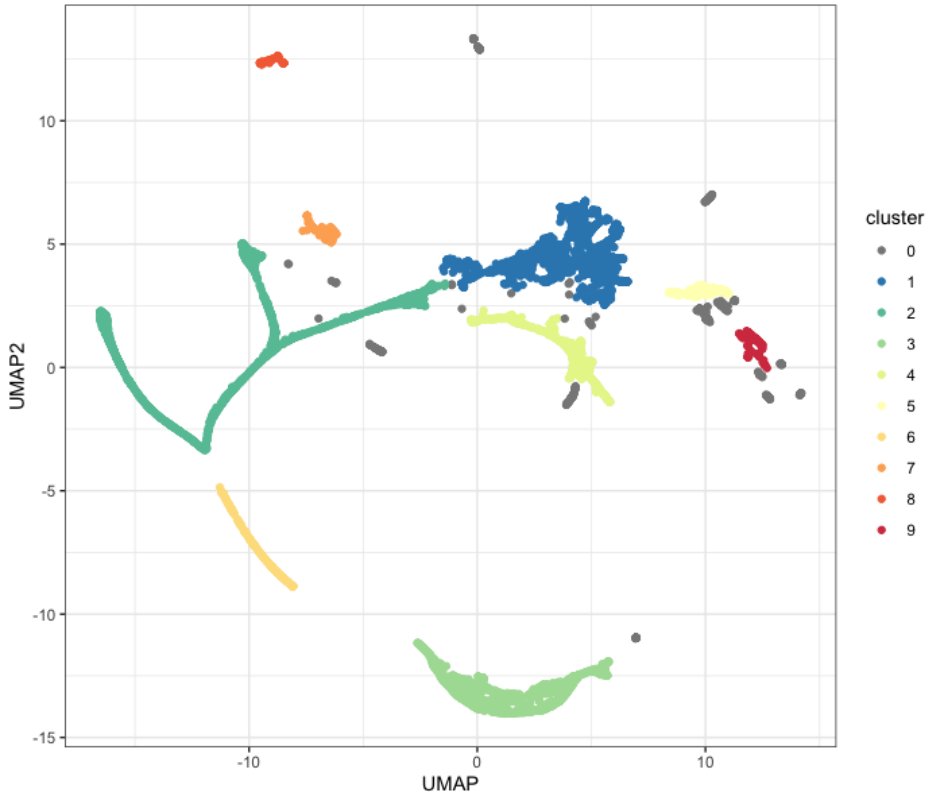


Figure 1.24: Graph showing the results from applying the DBSCAN clustering algorithm, with parameters minP ts = 4 and eps = 0.2963, to the UMAP projection of the data, after keeping the 9 biggest clusters and grouping the rest of the datapoints into cluster 0, shown in gray.

Cluster 6, composed of the string of dark yellow datapoints located between clusters 2 and 3 in Figure 1.24, shares a lot of the characteristics with cluster 3, with the main difference that the values of *duration_inside* of cluster 6 are smaller, with an average of 2851.0978 seconds, compared to the average of cluster 3 of 17358.9323 seconds. This fact is reflected in the parent node of both leaf nodes in the decision tree, where the varaible used to split the nodes is *duration_inside*.

Cluster 7, shown in light orange in Figure 1.24, contains datapoints which executed a large number of commands (*ncommands*) while having 0 outgoing connections (*n_connections*). Cluster 8, the dark orange points in Figure 1.24, is composed of the datapoints for which the *score_ip* had high values. This can be seen in the decision tree (Figure 1.25), where the preceeding rules of the cluster 8 leaf nodes spliitted the data with the *score_ip* variable. A similar group of data was identified by the k-means algorithm, where the datapoints of cluster 5 had high values of *score_ip*.

cluster_text	rows	n_tries_logins	words_user	words_pass	duration_time_login	n_connections	duration_inside	nfiles_down	nfiles_create	ncommands	score_ip
	Min.	1.0000	4.0000	0.0000	0.1219	0.0000	0.5859	0.0000	0.0000	0.0000	0.0000
	Max.	7.0000	7.0000	50.0000	84.5535	36.0000	26.1330	2.0000	0.0000	20.0000	16.0000
Cluster 1	1st Qu.	1.0000	4.0000	5.0000	0.7983	0.0000	1.6766	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	6.0000	1.4649	0.0000	3.1283	0.0000	0.0000	0.0000	0.0000
	Mean	1.0672	4.0033	6.0296	2.4312	0.1786	6.2534	0.0179	0.0000	0.3460	0.0819
	3rd Qu.	1.0000	4.0000	8.0000	2.3917	0.0000	9.6809	0.0000	0.0000	0.0000	0.0000
	Min.	1.0000	4.0000	0.0000	0.0844	0.0000	21.5844	0.0000	0.0000	0.0000	0.0000
	Max.	2.0000	7.0000	56.0000	66.2306	105.0000	403.5752	8.0000	50.0000	55.0000	41.0000
Cluster 2	1st Qu.	1.0000	4.0000	4.0000	0.2559	0.0000	33.8786	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	6.0000	0.3389	0.0000	94.5657	0.0000	0.0000	19.0000	0.0000
	Mean	1.0014	4.0028	5.8622	1.1623	0.7797	125.1175	0.9162	0.0251	25.8470	0.0388
	3rd Qu.	1.0000	4.0000	8.0000	0.7059	1.0000	200.5786	1.0000	0.0000	55.0000	0.0000
	Min.	1.0000	4.0000	0.0000	0.1695	0.0000	8768.0578	0.0000	0.0000	0.0000	0.0000
	Max.	1.0000	4.0000	8.0000	15.7856	22180.0000	621439.0016	0.0000	0.0000	0.0000	17.0000
Cluster 3	1st Qu.	1.0000	4.0000	0.0000	0.6801	27.0000	11648.4951	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	0.0000	1.0291	58.0000	11896.2669	0.0000	0.0000	0.0000	0.0000
	Mean	1.0000	4.0000	0.0360	1.1266	142.4711	17358.9323	0.0000	0.0000	0.0000	0.1690
	3rd Qu.	1.0000	4.0000	0.0000	1.2354	142.0000	12191.7274	0.0000	0.0000	0.0000	0.0000
	Min.	1.0000	4.0000	0.0000	0.1743	0.0000	0.6492	0.0000	0.0000	0.0000	0.0000
	Max.	2.0000	4.0000	2.0000	5.9272	3.0000	22.6252	1.0000	0.0000	4.0000	0.0000
Cluster 4	1st Qu.	1.0000	4.0000	0.0000	0.2533	0.0000	1.0711	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	0.0000	0.3269	1.0000	2.3166	0.0000	0.0000	0.0000	0.0000
	Mean	1.0033	4.0000	0.0722	0.7219	0.6937	6.1027	0.0011	0.0000	0.0613	0.0000
	3rd Qu.	1.0000	4.0000	0.0000	0.8583	1.0000	11.1629	0.0000	0.0000	0.0000	0.0000
	Min.	1.0000	4.0000	4.0000	0.1872	0.0000	2.3283	0.0000	0.0000	0.0000	0.0000
	Max.	1.0000	4.0000	6.0000	3.8258	1.0000	6.6535	0.0000	0.0000	1.0000	17.0000
Cluster 5	1st Qu.	1.0000	4.0000	5.0000	1.4179	1.0000	2.7841	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	5.0000	1.4703	1.0000	2.9774	0.0000	0.0000	0.0000	0.0000
	Mean	1.0000	4.0000	5.0000	1.4532	0.9847	3.2846	0.0000	0.0000	0.0022	0.0415
	3rd Qu.	1.0000	4.0000	5.0000	1.5317	1.0000	3.4615	0.0000	0.0000	0.0000	0.0000
	Min.	1.0000	4.0000	0.0000	0.2163	0.0000	417.6114	0.0000	0.0000	0.0000	0.0000
	Max.	2.0000	4.0000	11.0000	23.7594	1437.0000	8435.4604	1.0000	0.0000	55.0000	100.0000
Cluster 6	1st Qu.	1.0000	4.0000	0.0000	0.6406	16.0000	871.6473	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	0.0000	0.9407	120.5000	2050.0540	0.0000	0.0000	0.0000	0.0000
	Mean	1.0032	4.0000	1.0491	1.9476	213.9620	2851.0978	0.0032	0.0000	3.1297	0.6329
	3rd Qu.	1.0000	4.0000	0.0000	1.4618	314.2500	4334.4466	0.0000	0.0000	0.0000	0.0000
	Min.	1.0000	4.0000	0.0000	0.1547	0.0000	0.6647	0.0000	0.0000	10.0000	0.0000
	Max.	1.0000	4.0000	11.0000	9.3655	0.0000	22.8105	8.0000	0.0000	27.0000	0.0000
Cluster 7	1st Qu.	1.0000	4.0000	0.0000	0.2438	0.0000	10.7217	0.0000	0.0000	18.0000	0.0000
	Median	1.0000	4.0000	4.0000	0.4288	0.0000	11.0684	1.0000	0.0000	19.0000	0.0000
	Mean	1.0000	4.0000	3.4527	0.6629	0.0000	11.2213	1.3581	0.0000	18.9291	0.0000
	3rd Qu.	1.0000	4.0000	5.0000	0.6349	0.0000	12.3291	2.0000	0.0000	19.0000	0.0000
	Min.	1.0000	4.0000	0.0000	0.2605	0.0000	0.6710	0.0000	0.0000	0.0000	24.0000
	Max.	2.0000	4.0000	9.0000	8.8122	4.0000	45.8755	0.0000	0.0000	27.0000	100.0000
Cluster 8	1st Qu.	1.0000	4.0000	0.0000	0.3034	1.0000	1.1607	0.0000	0.0000	0.0000	41.0000
	Median	1.0000	4.0000	5.0000	0.3190	4.0000	1.8388	0.0000	0.0000	0.0000	41.0000
	Mean	1.0085	4.0000	2.8654	0.5561	2.8590	4.6378	0.0000	0.0000	0.2308	42.6752
	3rd Qu.	1.0000	4.0000	5.0000	0.3935	4.0000	4.0722	0.0000	0.0000	0.0000	41.0000
	Min.	1.0000	4.0000	5.0000	0.1431	0.0000	0.6224	0.0000	0.0000	0.0000	0.0000
	Max.	1.0000	4.0000	6.0000	1.4713	2.0000	2.8218	0.0000	0.0000	0.0000	0.0000
Cluster 9	1st Qu.	1.0000	4.0000	5.0000	0.3146	1.0000	1.1139	0.0000	0.0000	0.0000	0.0000
	Median	1.0000	4.0000	5.0000	0.5818	1.0000	1.3091	0.0000	0.0000	0.0000	0.0000
	Mean	1.0000	4.0000	5.0088	0.5270	0.9167	1.3472	0.0000	0.0000	0.0000	0.0000
	3rd Qu.	1.0000	4.0000	5.0000	0.6320	1.0000	1.6075	0.0000	0.0000	0.0000	0.0000

Table 1.12: Summary metrics calculated for all points of each of the clusters obtained with the DBSCAN algorithm.

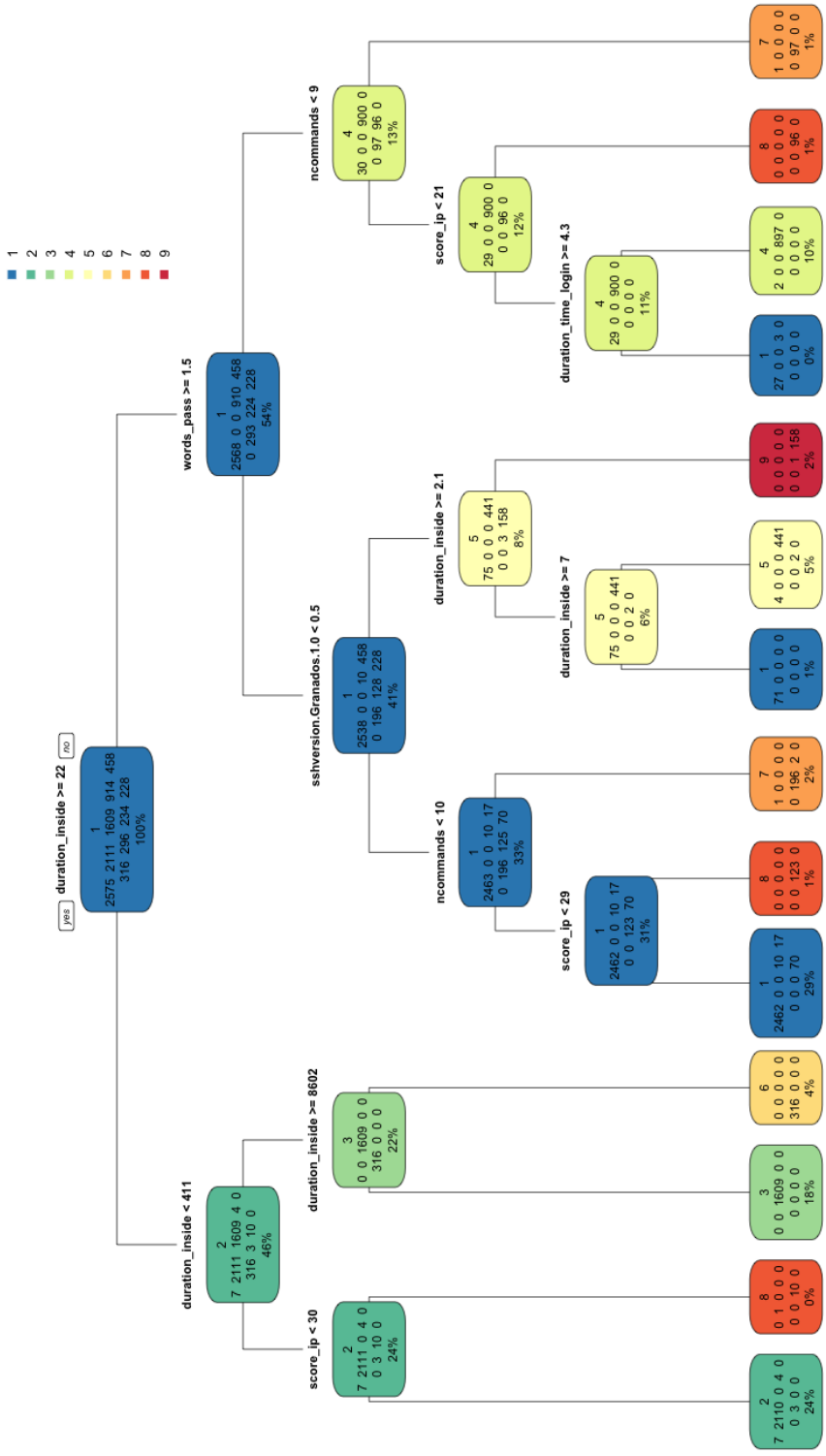


Figure 1.25: *Decision tree obtained using the results of the DBSCAN clustering algorithm.*

1.2.5 Remarks about the results in this chapter

In this chapter we introduced three different clustering algorithms and then used them with real data to be able to find patterns in the data collected by a honeypot. The first algorithm we used, k-means clustering algorithm, gave good results overall. It was able to identify 8 different clusters in the honeypot data. Most of the clusters had interpretable characteristics. Some clusters identified typical behaviour of attacks. For example, we found cluster 4 was only using the honeypot to create outgoing connections to other points. The second algorithm we used to characterise the data was the k-prototypes algorithm. This algorithm is an adaptation of the k-means algorithm, that enables it to analyse categorical and numerical variables. We included the *sshversion* categorical variable into the clustering analysis. The results obtained by k-prototypes were not as good as the ones outputted by k-means. Initially, when tuning the hyperparameter, k, the k-prototypes algorithm identified two clusters in the data. This, however, did not provide a lot of useful information about the data. We decided to discard these results and pick the second best result obtained during the hyperparameter tuning process. We trained a k-prototypes model, this time with 4 clusters. There were some similarities between the clusters found by the two algorithms, which tells us the algorithms were able to capture similar patterns in the data. The inclusion of the *sshversion* in the analysis did not give the expected results. We were not able to draw any useful conclusions with it. Lastly, we used the UMAP algorithm to reduce the dimensions of our data to two dimensions. One of the advantages of doing this is that it made the clustering analysis more intuitive. We used the DBSCAN algorithm to cluster the two-dimensional representation of the data. Initially the algorithm identified 25 different clusters. However, to simplify the analysis, we decided to keep a number of clusters which contained more than 90% of the data, and group the remaining points in an outlier cluster. This resulted in 9 different clusters. We were able to analyse the characteristics of each cluster and draw conclusions from them. Again, some of the clusters found by the DBSCAN algorithm were similar to the clusters formed by the k-means and k-prototypes algorithm. Overall, the DBSCAN algorithm provided with the best results. The clusters that were formed had distinct characteristics that were useful for describing the data.

We are going to conclude this chapter by mentioning some areas of opportunity and ideas that could further improve the performance of the clustering techniques that were covered, as well as some issues and failed attempts encountered during the execution of the analyses.

- During the data preprocessing stage we transformed the data by dividing each feature variable by its range. There exist other scaling alternatives that could improve the results, such as normalisation (scaling the variable to have values between 0 and 1) and standardisation (transforming the data to have zero mean and a standard deviation of 1), that could change the results of the clustering algorithms.
- During the experimentation stage, we performed the k-means algorithm without doing any data preprocessing, that is, no rescaling of the variables. The results turned out as expected: the variable with the largest range in the data, *duration_inside* dominated the results. The algorithm divided the data into two groups, one consisting of data points for which the variable *duration_inside* had small values and another with large values.
- A Principal Component Analysis (PCA) was performed on the dataset during the preprocessing stage, in an attempt to reduce the dimensionality of the data before applying the clustering algorithms. The results of the analysis were underwhelming as they showed that we would have to include 9 out of 11 principal components in order to explain a high percentage of the variability in the data. We concluded there was little benefit to be gained from using PCA.
- We used the Silhouette metric to evaluate the performance of the clustering algorithms, but there are many more metrics such as the Dunn (Dunn, 1973) and Davies-Bouldin (Davies and Bouldin,

1979) indeces, which favor other properties of the clusters. An analysis could be done to analyse and compare the results from using such indices.

- An more rigorous exploratory analysis could be performed on the data to understand its nature better. One could use this information to select more appropriate strategies the tackle the problems at hand.
- It could be beneficial to employ a clustering algorithm that does not segment the data with hard partitions. Such is the case of the *fuzzy k-means* algorithm, in which the membership of points to different clusters can vary from 0 to 1.
- *Ordering points to identify the clustering structure*, or OPTICS, is another density based algorithm. It has an advantage over DBSCAN that it can detect clusters in data of varying density. It could be worthwhile to experiment with this algorithm to see if interesting results are obtained.
- In this work we did not perform any validation analysis of the results obtained from the clustering algorithms. This is an important step, because we need to be able to measure the goodness of the partitions after clustering. More information about clustering validation techniques can be found in Chapter 23 of Aggarwal and Reddy (2014).
- The overall implementation of the system could be modified to include an online learning method, as to allow for dynamic modelling of network data, i.e. the ability to learn from new data as it arises. This, however, has its own challenges like the difficulty of retraining the algorithm used for the characterisation to adapt to incoming data.

Chapter 2

Forecast prediction of number of attacks

2.1 Overview of forecasting algorithms

In this section we will introduce the theoretical concepts needed to obtain the forecasts of a time series using machine learning algorithms. We explore the use of two algorithms, random forests (Breiman, 2001), and generalised additive models (Hastie and Tibshirani, 1986), to make forecasts. Specifically, we are interested in making predictions of the number of attacks a honeypot will receive in a day based on past data. From a cybersecurity point of view, it is important to understand to what extent, the number of cyber attacks can be predicted. This allows security experts to allocate the sufficient resources and strategies to counter the attacks. Although there are more classical approached used in time series forecasting, like ARIMA models, in this work we were interested in exploring the random forests algorithm and generalised additive models to see how well they were able to predict the number of attacks.

2.1.1 Forecasting with Random Forests

To explain how random forests work, first we need to introduce the concept of decision trees. A *decision tree* is a supervised machine learning technique for solving both classification and regression problems. They help in identifying the relationship among datapoints in a dataset by constructing tree-like structures. These tree-like structures are used to make accurate predictions about unseen data. The dataset is split into multiple subsets, thereby resulting in each decision node branching to more decision nodes. Decision trees are very popular because the algorithm and its output are very easy to explain; they are easy to understand as it resembles human decision-making. A trained decision tree can be displayed graphically which makes it easy to interpret. Also, decision trees can handle continuous as well as categorical variables, without the need to create dummy variables for the latter.

As mentioned earlier, decision trees can be applied to both regression and classification problems. When they are used for regression, they are known as regression trees, and, when used for classification,

they are known as classification trees. However, for the sake of simplicity, we are going to focus only on regression trees. For a whole explanation and examples of classification trees see James et al. (2013).

There are many methodologies for constructing decision trees but the most popular one is the classification and regression tree (CART) algorithm proposed in Breiman et al. (1984). A basic decision tree partitions the training data starting from a root node for the whole learning sample and ending with relatively homogeneous nodes, and then fits a simple constant in each node. In regression trees, the value of this constant is the average of the target values in the node. The nodes are formed by making recursive binary partitions using an *if-then-else* strategy about each variable. This is done a number of times until a suitable stopping criteria is satisfied. After all the partitioning has been done, the regression model predicts the output based on the average target values of all datapoints that fall in that node. An example of a regression tree can be seen in Figure 2.1.

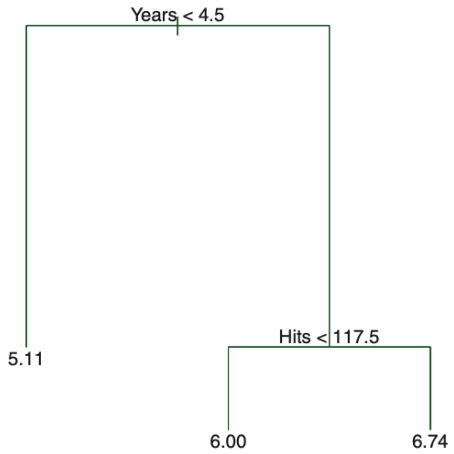


Figure 2.1: A regression tree was trained for predicting the log salary of a baseball player, based on the number of years active and the hits he made in the previous year. At each node, the label indicates the rule the decision tree is using for the split. Source: James et al. (2013)

We refer to the first node at the top of the tree as the root node (this node contains all of the training data). The final nodes at the bottom of the tree are called the terminal nodes or leaf nodes. Every node in between is referred to as an internal node. The connections between nodes are called branches (see Figure 2.2).

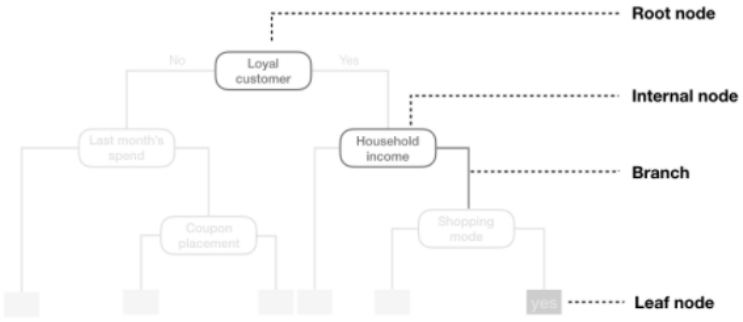


Figure 2.2: Example of a decision tree with definitions for each type of node. Source: Hands-On Machine Learning with R.

The regression tree needs to find the best variable to partition the data into subgroups at each node. Let $D = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ be a dataset of N datapoints, p features (variables) and a target variable, with $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})$. The objective at each note is to find the variable to partition the remaining data into one of the two regions, R_1 and R_2, such that the overall error between the actual target y_i and the mean target values within each region, $\hat{y}_{R_j}$ for $j \in \{1, 2\}$, minimise the total sum of squared errors given by

$$SSE = \sum_{i \in R_1} (y_i - \hat{y}_{R_1})^2 + \sum_{i \in R_2} (y_i - \hat{y}_{R_2})^2.$$

Having found the best feature-split combination, the data are partitioned into two regions and the splitting process is repeated on each of the two regions. This process is continued until a stopping criterion is met. Common stopping criteria are tree depth and number of terminal leaves.

Decision trees have the property of being a low-bias, high-variance algorithm. Often a small change in the data can result in a very different series of splits, making interpretations somewhat dangerous. The major reason for this instability is the hierarchical nature of the process: the effect of an error in the top split is propagated down to all of the splits below it. Therefore, decision trees would benefit from a method of reducing variance that could greatly improve upon the predictive accuracy of the technique. One salient approach to reducing variance of decision trees is to train a bunch of decision trees on different random subsets of the training data, by sampling with replacement. This approach is called bagging.

The main idea behind *bagging* or *bootstrap aggregating* (Breiman, 1996) is that for a given dataset $Z = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$, we obtain a collection of bootstrap samples $\mathbf{Z}^{*b}, b = 1, \ldots, B$, and we fit a model for each sample. In our context, the models being calculated with each bootstrap sample are regression tree models. After calculating the B models, we obtain B predictions $\hat{f}^{*b}(x)$ and calculate the bagging estimate as

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x). \tag{2.1}$$

Random forests are a modification of bagged decision trees, which work by building a large collection of uncorrelated trees to improve the predictions. They are built using the same fundamental principles as decision trees. The technique of bagging introduces a random component into the tree building process by fitting many trees on bootstrapped copies of the training data, and then aggregates the predictions across all the trees; this aggregation reduces the variance of the overall procedure and results in improved predictive performance. Bagging, however, also introduces correlation between the trees: all the bagged trees tend to make the same splits at each node because they all share the same features. This makes all these trees look very similar hence it increases their correlation. To fix this issue, the random forest algorithm chooses m features at random to perform the split at each node, allowing for individual trees in the random forest to be uncorrelated James et al. (2013). A typical value for the number of m is $\sqrt{p}$, that is, the square root of the number of variables; however, for the regression setting, it is recommended that the default value for m is $\lfloor p/3 \rfloor$, and to have a minimum node size of 5, see Hastie et al. (2001). Reducing the value of m will reduce the correlation between any pair of trees, and hence reducing the variance of the average.

Similarly to decision trees, random forests can be used for both classification and regression tasks.

Given that we want to make forecasts, we will focus on random forests for regression. Algorithm 2.1 shows the steps for the random forest regression algorithm.

Algorithm 2.1: Random forest for regression (Hastie et al., 2001).

1. For $b = 1, 2, \ldots, B$, repeat:
   (a) Draw a bootstrap sample Z^* of size N from the training data.
   (b) Train a random forest tree T_b with Z^*, and for each node of the tree, repeat the following steps until the minimum node size $n_{\min}$ is reached:
      i Select m predictor variables at random, with probability $1/p$.
      ii Select the best variable and split from all the sampled variables.
      iii Split the node into two.
2. Output the collection of decision trees $\{f_b\}_1^B$
   To make a prediction for a given point x, calculate $\hat{f}_{rf}^B(x) = \frac{1}{B}\sum_{b=1}^{B} f_b(x)$

Random forests obtain new predictions by averaging the outputs of all the trees evaluated at the new point x.

Out of Bag Samples

Bagging effectively causes about 1/3 of the original training data to be excluded from each individual decision tree (Hastie et al., 2001). This excluded data is referred to as the out of bag (OOB) samples. An out of bag error can be calculated using the OOB samples: we can predict the response for the ith datapoint using each of the trees in which that datapoint was OOB. This will yield around $B/3$ predictions for the ith datapoint. In order to obtain a single prediction for the ith datapoint, we can average these predicted responses. This leads to a single OOB prediction for the ith datapoint. An OOB prediction can be obtained in this way for each of the N datapoints, from which the overall OOB mean square error can be computed. This is called the OOB error.

Variable importance

Random forests use the OOB samples to construct a variable importance metric, to measure the prediction strength of each variable. When the bth tree is grown, the OOB samples are passed down the tree, and the prediction accuracy is recorded. Then the values for the jth variable are randomly permuted in the OOB samples, and the accuracy is again computed. The decrease in accuracy as a result of this permuting is averaged over all trees, and is used as a measure of the importance of variable j in the random forest. A variable that is important to the model will have a large impact in the accuracy in the model.

2.1.2 Forecasting with GAMs

In this section we are going to introduce the *generalised additive models*, or GAMs, that will be used in the forecast predictions of the data. Before we do so, we are going to make a brief overview of simpler models, the linear model and generalised linear models, which are the precursors of the GAMs.

Linear model

The simplest regression model there is the linear model, which predicts a response as a sum of linear predictors. It is one of the most popular regression models given its simplicity and intuitive interpretation of the results. Even though it is not as sophisticated compared more modern statistical learning algorithms, it serves as a good starting point for more advanced approaches. In a regression model, the relationship between the target variable Y and p features (covariables) $\mathbf{X} = (X_1, \ldots, X_p)$ is given by

$$Y = \mu(\mathbf{X}) + \epsilon$$

where $\mu(\mathbf{X}) = \mathbb{E}[Y|\mathbf{X}]$ is the conditional expected value of the response Y given $\mathbf{X}$, and ϵ is a measure of the error. The most basic regression model is the linear regression model which assumes a linear relationship between the target variable and the predictors, defined as

$$\mu(\mathbf{X}) = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p + \epsilon$$

where $\beta = (\beta_0, \beta_1, \ldots, \beta_p)$ are the parameters of the linear model to be fitted, and now the error ϵ is assumed to be normally distributed and with mean zero and constant variance σ^2. As stated before, the model is widely used given its simplicity and easy interpretation of the parameters, however, the model makes several strong assumptions, like the linear relationship between the target variable and the predictors, and the assumption of homoscedasticity of the errors. This limits the ability of the model to capture more complex relationships between the data, and thus generates the need for other algorithms that provide greater flexibility.

Generalised linear model

Generalised linear models (Nelder and Wedderburn, 1972), or GLMs, are an extension of the simple linear model, capable of modelling a target variable without the need for it to follow a normal distribution. They are modelled as

$$g(\mu(\mathbf{X})) = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p \qquad Y \sim \mathrm{EF}(\mu_i, \phi)$$

and allow Y to follow a distribution from the exponential family (e.g. binomial, normal, Poisson, gamma, etc.), where g is a *link function*, and ϕ is a scale parameter. Table 2.1 contains the most common link functions.

Link function g	$\mu = g^{-1}(\eta)$	Exponential family	Scale ϕ
$g(\mu)$	$\mu = \eta$	Normal	σ^2
$g(\mu) = \log\left(\frac{\mu}{1-\mu}\right)$	$\mu = \frac{\exp}{1+\exp}$	Binomial	1
$g(\mu) = \log(\mu)$	$\mu = \exp(\eta)$	Poisson	1
$g(\mu) = \frac{1}{\mu}$	$\mu = \frac{1}{\eta}$	Gamma	$\frac{1}{\nu}$

Table 2.1: Common link functions used in GLMs.

Estimation of GLMs

One way of finding the estimates of the parameters $\beta = (\beta_0, \beta_1, \ldots, \beta_p)$ is by using the Fisher scoring algorithm (see Wood (2006) for more details) outlined in Algorithm 2.2. The basic idea of the algorithm is to calculate the estimates of the parameters $\hat{\beta}$ using maximum likelihood estimation (MLE). However, because the expressions found by MLE are non-linear, we need to make use of a numerical method that iteratively finds the estimates of the parameters. Such algorithm is called iteratively re-weighted least squares (IRLS).

Algorithm 2.2: Local scoring algorithm.

1. *Initialise the parameters*: $\hat{\beta}_0^0 = g(\sum_{i=1}^{N} \frac{Y_i}{N})$; $\hat{\beta}_1^0 = \cdots = \hat{\beta}_p^0 = 0$
2. *Update*:
   (a) Compute, for $i = 1, \ldots, N$, the variable
   $$Z_i = \eta_i(\hat{\boldsymbol{\beta}}^0) + g'(\hat{\mu}_{0i})(Y_i - \hat{\mu}_{0i}) = \eta_i(\hat{\boldsymbol{\beta}}^0) + \frac{(Y_i - \hat{\mu}_{0i})}{\hat{\mu}_{0i}}$$
   (b) Compute the weights
   $$W_i = \frac{1}{g'(\hat{\mu}_{0i})^2 \mathrm{Var}(Y_i)} = \hat{\mu}_{0i}$$
   where $\eta_i(\hat{\boldsymbol{\beta}}^0) = \hat{\beta}_0^0 + \hat{\beta}_1^0 X_{i1} + \cdots + \hat{\beta}_p^0 X_{ip}$
   and $\hat{\mu}_{0i} = g^{-1}(\eta_i(\hat{\boldsymbol{\beta}}^{\mathbf{0}}))$
   (c) Fit a weighted linear model to obtain the estimated updated parameters
   $$\hat{\boldsymbol{\beta}} = (\hat{\beta}_0, \cdots, \hat{\beta}_p)^t = (\mathbb{X}^t \mathbb{W} \mathbb{X})^{-1} \mathbb{X}^t \mathbb{W} \boldsymbol{Z}$$
   where
   $$\mathbb{X} = \begin{pmatrix} 1 & X_{11} & \cdots & X_{1p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & X_{n1} & \cdots & X_{np} \end{pmatrix} \text{ and } \mathbb{W} = diag(W_1, \ldots, W_n)$$
   (d) Compute the convergence criterion
   $$\frac{\|\hat{\boldsymbol{\beta}} - \hat{\boldsymbol{\beta}}^0\|}{\|\hat{\boldsymbol{\beta}}^0\|} \leq \varepsilon$$
   where $\|\cdot\|$ is the euclidean norm and ε is a small error threshold.
3. Repeat step 2 replacing $\hat{\beta}^0$ by $\hat{\beta}$ obtained in step 2.c until convergence criterion is below some small threshold ε.

Generalised additive model

A generalised additive model (Hastie and Tibshirani, 1986), or GAM, is a generalisation of a GLM to incorporate non-linear forms of the predictors; it is composed of a sum of smooth functions of the predictors instead of or in addition to the standard linear predictor effects. In general, they have the following structure

$$g(\mu) = \beta_0 + f_1(X_1) + \ldots + f_p(X_p)$$

where f_j are the smooth (non-parametric) functions of the predictors, allowing for a more flexible relationship between them and the target variable. In general, GAMs have the interpretability advantages of GLMs where the contribution of each independent variable to the prediction is clearly encoded. However, it has substantially more flexibility because the relationships between the target variable and the predictors are not assumed to be linear. In fact, we don't have to know a priori what type of predictive functions we will eventually need. From an estimation standpoint, the use of regularised, nonparametric functions avoids the pitfalls of dealing with higher order polynomial terms in linear models. From an accuracy standpoint, GAMs are competitive with popular learning techniques.

Estimation of GAMs

To estimate a GAM model we can adapt the algorithm we used for the GLMs in Section 2.1.2. Especifically, in step 2.c, where instead of fitting a weighted linear model, the algorithm will fit a weighted additive linear model. The steps for this modified procedure are outlined in Algorithm 2.3.

In step 2.c of Algorithm 2.3 we need a way to fit the weighted additive model. This is usually done with *penalised regression splines*, where each of the functions f_j is represented in such a way that it becomes a linear model, which can be accomplished by selecting some *basis functions* so that they can be expressed as

$$f_j(x_j) = \sum_{k=1}^{K_j} b_{jk}(x_j)\beta_{jk}$$

where $b_{jk}(\cdot)$, $k = 1, \ldots, K_j$, are the basis functions, K_j is the number of knots, and $\beta_{jk}(\cdot)$ are unknown parameters. The additive model can be viewed as the following linear model

$$\eta_i = \sum_{k=1}^{K_1} b_{1k}(x_{1i})\beta_{1k} + \ldots + \sum_{k=1}^{K_p} b_{pk}(x_{pi})\beta_{pk} = \boldsymbol{X_i}\boldsymbol{\beta} \tag{2.2}$$

where

$$\boldsymbol{X_i} = \begin{pmatrix} b_{11}(x_{1i}) & \cdots & b_{1K_1}(x_{1i}) & \cdots & b_{p1}(x_{pi}) & \cdots & b_{pK_p}(x_{pi}) \end{pmatrix}$$

and

$$\boldsymbol{\beta} = \begin{pmatrix} \beta_{11} & \cdots & \beta_{1K_1} & \cdots & \beta_{p1} & \cdots & \beta_{pK_p} \end{pmatrix}.$$

The idea of using basis functions to estimate f_j introduces a new problem: the selection of the number of knots. An approach to solve this problem is to keep the basis dimension fixed at a size a little

Algorithm 2.3: Local scoring algorithm for GAMs.

1. *Initialise the parameters*: $\hat{\beta}_0^0 = g(\sum_{i=1}^{N} \frac{Y_i}{N})$; $\hat{\beta}_1^0 = \cdots = \hat{\beta}_p^0 = 0$
2. *Update*:
   (a) Compute, for $i = 1, \ldots, N$, the variable
   $$Z_i = \hat{\eta}_i^0(\hat{\boldsymbol{\beta}}^0) + g'(\hat{\mu}_{0i})(Y_i - \hat{\mu}_{0i}) = \hat{\eta}_i^0(\hat{\boldsymbol{\beta}}^0) + \frac{(Y_i - \hat{\mu}_{0i})}{\hat{\mu}_{0i}}$$
   where $\hat{\eta}_i^0(\hat{\boldsymbol{\beta}}^0) = \hat{\beta}_0^0 + \hat{f}_1^0(X_{i1}) + \cdots + \hat{f}_p^0(X_{ip})$
   and $\hat{\mu}_{0i} = g^{-1}(\hat{\eta}_i^0(\hat{\beta}^{\mathbf{0}}))$
   (b) Compute the weights
   $$W_i = \frac{1}{g'(\hat{\mu}_{0i})^2 \mathrm{Var}(Y_i)} = \hat{\mu}_{0i}$$
   (c) Fit an additive model weighted by $\mathbb{W}$ to $\boldsymbol{Z}$, in order to obtain, for $i = 1, \ldots, N$, the updates of the estimated parameters
   $$\hat{f}_1(X_{i1}), \ldots, \hat{f}_p(X_{ip})$$
   and
   $$\hat{\eta}_i = \hat{\beta}_0 + \hat{f}_1(X_{i1}, \ldots, \hat{f}_p(X_{ip}))$$
   (d) Compute the convergence criterion
   $$|\hat{\eta}_i - \hat{\eta}_i^0| \leq \varepsilon$$
   where ε is a small error threshold.
3. Repeat step 2 replacing $\hat{\eta}_i^0$ by $\hat{\eta}_i$ obtained in step 2.c until the convergence criterion is below some small threshold ε.

larger than it is believed could reasonably be necessary, and instead control the model's smoothness by adding a wiggliness penalty to the least squares fitting objective. Thus, the coefficients $\boldsymbol{\beta}$ in Equation 2.2 can be estimated minimising the penalised sum of square residuals

$$\sum_{i=1}^{n} w_i(Z_i - \boldsymbol{X_i}\boldsymbol{\beta})^2 + \lambda_1 \int (f''(X_1))^2 dx_1 + \cdots + \lambda_p \int (f''(X_p))^2 dx_p$$

where $\lambda_j \geq 0$, $j = 1, \ldots, p$, are the smoothing parameters that penalise the flexibility introduced by the number of basis. If the smoothing parameter is chosen too high, the estimations of f_j will be over smoothed, and if it is too low the estimations will be under smoothed. Common criteria to select the smoothing parameters are *generalised cross validation score*, or GCV and the *unbiased risk estimator*, or UBRE (Wood, 2006).

2.1.3 Evaluating forecast accuracy

To measure the performance of the forecasts we need to introduce some measures that will allow us how good the predictions of out model are. For this we are going to work with forecast errors, which are the difference between an actual observed value and its forecast. Let $\hat{y}_t$ be the estimation of Y_t using the training data set, then some of the most common measures used to summarise the forecast errors for the time windows $i = \{1, \dots, m\}$ are the following:

- *Mean absolute error (MAE)* - Calculates the mean absolute residual between the predictions and the true values
$$MAE = \frac{1}{m}\sum_{j=1}^{m} |y_{t+j} - \hat{y}_{t+j}|.$$

- *Root mean square error (RMSE)* - This metric calculates the mean of the squared difference between the predicted values and the true values, then takes the square root. This makes the values have the same scale as the outcome variable. Because the errors are squared, RMSE is more sensitive to outliers than MAE
$$RMSE = \sqrt{\frac{1}{m}\sum_{j=1}^{m}(y_{t+j} - \hat{y}_{t+j})^2}.$$

- Mean percentage error (MPE) This measure gives a percentage score of how forecasts deviate, on average, from the actual values
$$MAPE = 100\frac{1}{m}\sum_{j=1}^{m}\left|\frac{y_{t+j} - \hat{y}_{t+j}}{y_t}\right|.$$

2.2 Application to real data

Having introduced the theoretical background for the forecasating analysis, we are going to proceed to the practical analysis using real data. The data we are going to analyse contains daily counts of the number of attacks made to a Cowrie honeypot of the period between March 26th, 2018, and March 31st, 2019, containing 371 datapoints. In Figure 2.3 we see the graph of the series, where we can note it has a positive trend, with two big increases in the number of attacks, one occurring after the second week of February, 2019, and the other increase taking place after the second week of March, 2019. Although it is not always the case, it is normal to see a positive trend when analysing a time series of the number of attacks in honeypots. This happens because usually, the longer the honeypot is exposed, the more time the attackers have to discover and target the honeypot. Also, once it is discovered, the attacker may share the information of the honeypot with other attackers. The functions used in random forests analysis are from the *randomForest* library in R. For training the GAMs we used the functions in the *mgcv* library.

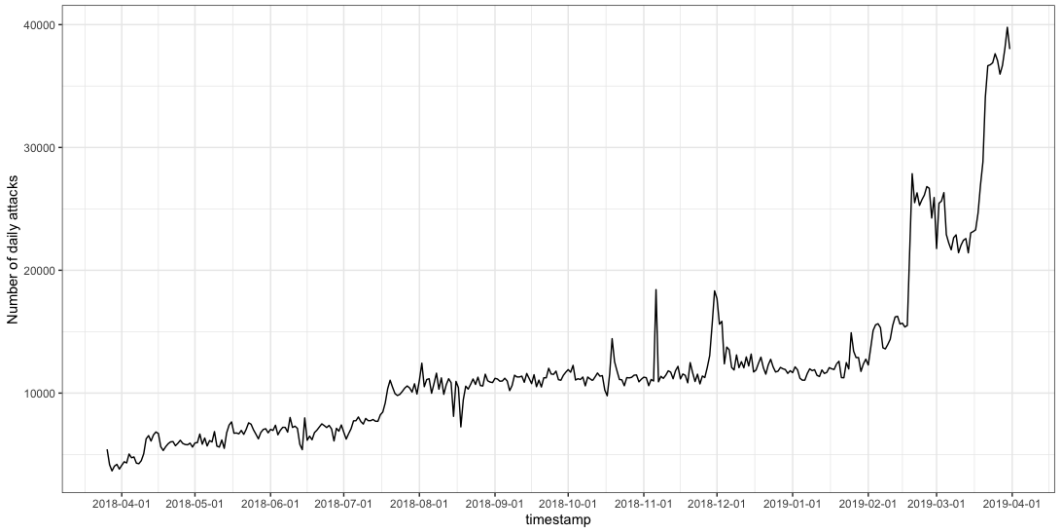


Figure 2.3: Graph of sequential data of the number of daily attacks made to a Cowrie honeypot.

Cross validation strategy

One way of measuring the accuracy of any supervised learning algorithm is to divide all the data into a training set and a test set. Usually, the algorithm is trained with just the training set, and its performance is later measured by predicting new values and comparing them to the test set. However, this particular way of measuring the performance can be misleading if the algorithm overfits or underfits the data, producing particularly bad predictions for the test set. One way to countermeasure this is by following a cross validation approach, as introduced in Section 1.2.2, this way we can train and test the performance of the algorithm many times, and then calculate an average of the performances.

Unlike the k-fold cross validation method used in Section 1.2.2 for clustering, where the splits were constructed by randomly sampling the data, in the context of time series we need make use of a *roll-forward* validation approach. This method involves selecting an initial training set starting at the origin

of the series $\{y_1, \ldots, y_{t1}\}$, and then selecting the following m points as the test set $\{y_{t1+1}, \ldots, y_{t1+m}\}$. For the second split, the training set will be $\{y_1, \ldots, y_{t2}\}$, where $t2 > t1$, and the corresponding test set will be $\{y_{t2+1}, \ldots, y_{t2+m}\}$. Using this technique, more splits are constructed until all the data in the series has been used. Figure 2.4 illustrates the main idea of splitting the data by using roll-forward validation approach.

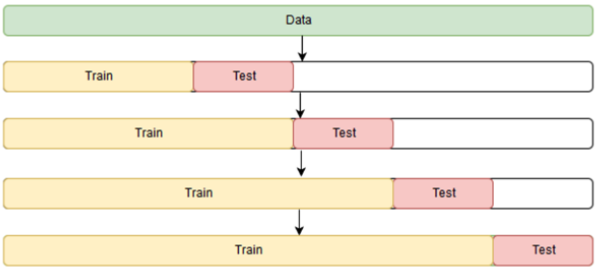


Figure 2.4: Diagram illistrating the *roll-forward* validation approach. Source Raschka (2015).

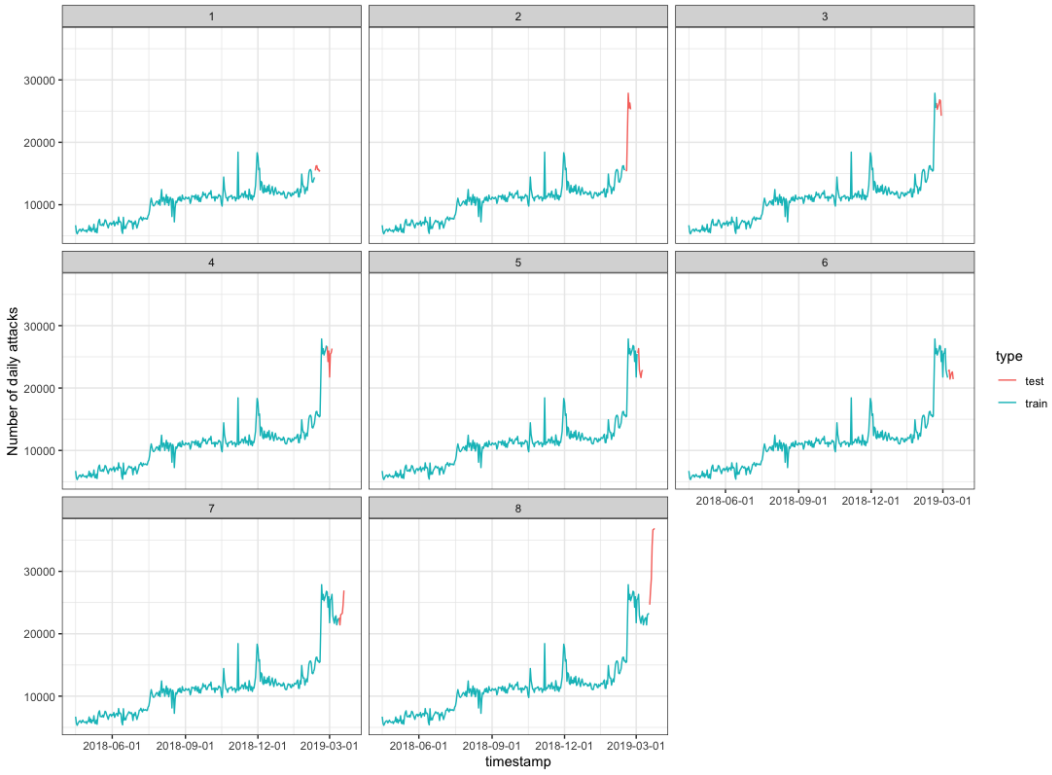


Figure 2.5: Plots of all the samples calculated for the roll-forward cross validation approach. Each sample consists of a training and a test set.

In order to follow this approach, first we reserved the last 7 observations of the series. These 7 points will be used until the end of the analyses to test the final model. This way we can test the models

with data that was never used for training them. Then, we divided the remaining 364 observations into 8 samples, where each sample is made of a training and test set. Because we are interested in predicting the number of attacks in the following 7 days, the test set in each split is of size 7. The plots of the 8 samples are shown in Figure 2.5, where the data considered for the training set in each sample is shown in green, and the test set in red.

2.2.1 Forecasting with Random Forests

The first algorithm we used to analyse the time series data is random forests algorithm, introduced in Section 2.1.1. The first step in the analysis was to augment the number of variables by doing some feature engineering. We added some variables we believed could be relevant to the analysis. Given the that we are dealing with daily data, we thought it would be useful to add the variables listed in Table 2.2. We introduced a number of lags, in case the series depends on past values. Lags 1, 2, and 3, were added so the algorithm can learn on data from the last 3 days. We also added a 7-day lag to add a weekly component, and a 14-day lag for a 2-week component.

Name of the variable	Description
year	Year, as an integer
month	Month, as an integer
day	Day of the month, as an integer
wday	Day of the week, as an integer, starting on Sunday
week	Number of the week in the year.
lag1	Lag of number of attacks with respect to the previous day
lag2	Lag of number of attacks with respect to the previous 2 days
lag3	Lag of number of attacks with respect to the previous 3 days
lag7	Lag of number of attacks with respect to the previous 7 days
lag14	Lag of number of attacks with respect to the previous 14 days

Table 2.2: Variables added to the data.

Given that the number of datapoints is of medium size (371), and the number of variables is 10, we decided to train a random forest for each possible combination of variables, starting with models containing a single variable, for example, models only containing *wday* or *lag1*, and finishing with a model containing all the variables. This resulted 1023 combinations of variables. Additionally, we also needed to tune the hyperparameters used by the random forest: *nodesize*, the minimum number of datapoints needed in each node, and *mtry*, the number of variables randomly sampled as candidates at each split. The strategy we followed was to create a grid made of combinations of some values of hyperparameters. For *nodesize* we considered a sequence of values between 5 and 50, in increments of 5, i.e., $nodesize \in \{5, 10, 15, \ldots, 50\}$. For *mtry* a sequence of values between 1 and 8 were considered, i.e., $mtry \in \{1, 2, \ldots, 8\}$. We obtained $10 \times 8 = 80$ combinations of possible values for the hyperparameters. Next, we trained $1023 \times 80 = 81840$ different random forests using the roll-forward validation approach, described in 2.2. That is, we trained one random forest per combination of variables per combination of hyperparameters. or each of the 8 samples, the models were trained and then used to predict data for a 7-day horizon. We used the test set of each sample to measure the performance of the models. This was computationally expensive, but given the low amount of datapoints making the computations feasable, we decided to use brute force to find the best combination of variables/hyperparameters.

	Model	RMSE	MAE	MAPE
1	n ∼ day + lag1 + lag2 + month + week + year	3423.610	2937.089	11.11969
2	n ∼ week	3481.160	2978.589	11.14853
3	n ∼ day + lag1 + lag7 + month + week + year	3505.215	3042.304	11.57367
4	n ∼ day + lag1 + lag2 + lag7 + month + week	3523.416	3042.107	11.55285
5	n ∼ day + lag1 + lag2 + lag7 + month + year	3533.367	3079.821	11.76944

Table 2.3: Top 5 models obtained after training 1023 different random forest models with different combination of variables, shown formula notation used by the `randomForest` function.

The top 5 models, ordered by the *RMSE* measure, are shown in Table 2.3. The best model predicts the number of daily attacks using the *day*, *month*, *week* and *year* variables. We see that it includes two lags, $lag1$ and $lag2$ in the model. This is the model which minimised the *RMSE*, MAE and $MAPE$ measures. The value of the $MAE = 2937.089$, which means that on average, the predictions deviated from the true value by approximately 2937 attacks. The *RMSE* is a little bigger with value of 3423.61, because it penalises the differences more severely. The $MAPE = 11.12$, which means that on average, the predictions were off by 11.12%. The hyperparameters related to the best model were $nodesize = 10$ and $mtry = 2$.

It is curious to see that the first 5 models included, more or less, the same variables, with exception to model two, which is comprised of the *week* variable. Most of the models included 2 or 3 lag variables, and a combination of the variables *day*, *week*, *month*, and *year*. None of the best models included the variables $lag3$, $lag14$ or *wday*. The second best model, $n \sim week$, only uses the variable *week* to make future predictions. This model is predicting future values by averaging the last values of the variable *week* that were used in the training data.

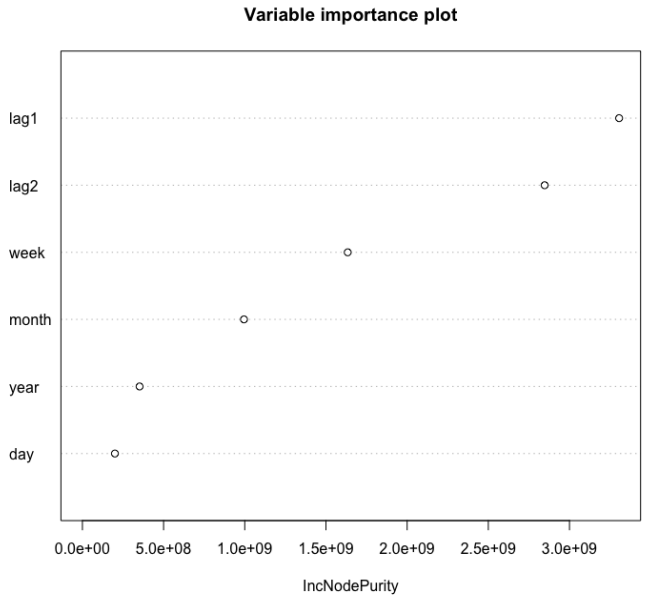


Figure 2.6: Variable importance plot of the random forest model.

Given the previous results, a random forest model was trained using the variables *dat*, *lag*2, *month*, *week*, and *year*, with values of the hyperparameters $nodesize = 10$ and $mtry = 2$. The data used for training the model was the training set, i.e., all the data minus 7 observations that were reserved as the test set. In Figure 2.6 we can see the variable importance plot, where the variables are rated by measuring the increment in node purity when the variable is included in the decision tree. The most important variables according to the algorithm are the first two lags, *lag*1 and *lag*2, and the *week* variable.

We calculated the predictions and compared them to the test set, composed of the last 7 observations of the series. These observations have never being used in the training of the model, as we wanted to test the performance with unseen data. The predictions are shown in Table 2.4 and are also depicted in Figure 2.7.

	Prediction	True value	Date
1	35558	37631	2019-03-25
2	35534	37075	2019-03-26
3	35534	35966	2019-03-27
4	35534	36672	2019-03-28
5	35462	38048	2019-03-29
6	35446	39783	2019-03-30
7	35446	38011	2019-03-31

Table 2.4: Predictions obtained for the last 7 days of the series.

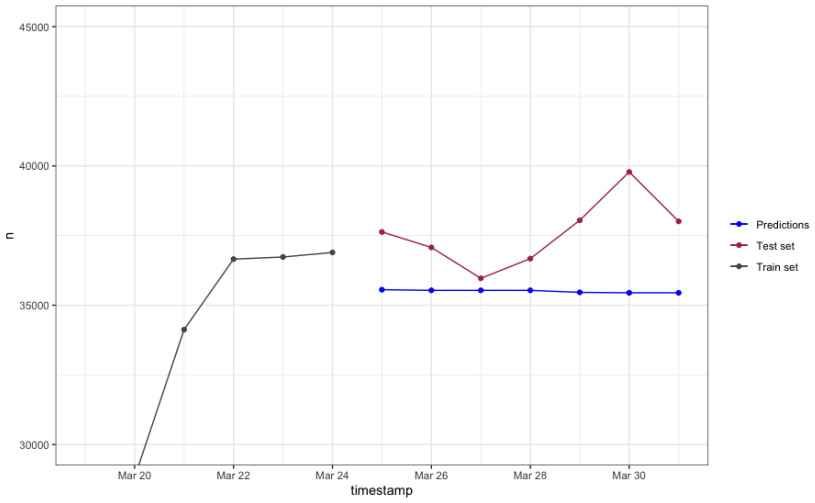


Figure 2.7: Graph of predicted values for the next 7 days, using the random forest algorithm. The original series is shown in black, and the predictions are in blue.

We calculated the accuracy metrics comparing the predictions with the observations to the test set to see how well the algorithm was able to generalise. The performance metrics can be seen in Table 2.5. The test *MAE* tells us that, on average, the predictions were off from the true values by 2096. The test *RMSE* is larger, because it penalises the differences more severely, with a value of 2397.316. The test *MAPE*= 5.49, which suggests the predictions were, on average, off by 5.49%.

RMSE	MAE	MAPE
2397.316	2096	5.487983

Table 2.5: Accuracy metrics calculated with the predictions compared with the data from the test set.

The test *MAPE* of 5.49% is acceptable and suggests the algorithm was able to generalise well. However, we have to note that all the predictions made by the decision tree are smaller than the true observations. This is more evident when looking at Figure 2.7. If we look at all the graph of the complete series (Figure 2.3), we notice that the last 7 points, which were used as the test set, are the points with the biggest values in the series. This presents a problem to the random forest algorithm, because of the way it learns from the data. At best, the algorithm can predict an average of training values it has seen before. Because the values of the test set are greater than those of the training set, the random forest will predict an average of values it has seen before, as it is not able to make extrapolations of the data.

2.2.2 Forecasting with GAMs

We continue the analysis with the GAM algorithm in the context of forecasting. As we already know, the series we want to analyse has a clear positive trend. In order for the GAM model to be able to capture the trend and seasonality, we are going to introduce a time index variable $t = 1, 2, 3, \ldots, T$. The inclusion of such variable has also been done in similar work done by Dominici et al. (2002) and Tobías and Saez (2004). The variables that are going to be included in the model are *lag1*, *day*, and *t*. We decided to use these particular variables because the cybersecurity team at Gradiant had already trained a successful GAM model to forecast the number of daily attacks received by a honeypot. So we were mainly interested to improve the existing model by focusing our efforts in tuning the number of basis of the smoothing functions. For the rest of the analyses in this section we used the following model

$$g(\mu) = \beta_0 + f_1(lag1) + f_2(day) + f_3(t),$$

where $g(\cdot)$ is the log link function, given the assumption that the target variable, y, which measures the number of attacks in a day, follows a Poisson distribution; β_0 is the intercept of the model, and f_1, f_2, and f_3 are the smoothing non-linear functions to be adjusted by the algorithm. The model can also be expressed in the following way

$$\ln(\mu) = \beta_0 + f_1(lag1) + f_2(day) + f_3(t). \tag{2.3}$$

Given that we want to know the optimal number of basis for the representation of the smoothing functions, a grid was constructed containing the combinations of the possible number of basis for each function. The number of basis considered for the variable $lag1$ is the sequence starting from 20 to 35 by increments of 3, i.e. $Nbasis_{lag1} = \{20, 23, 26, \ldots, 35\}$. Likewise, we considered the following sequences for variables *day* and *t*, $Nbasis_{day} = \{15, 18, 21, \ldots, 30\}$ and $Nbasis_t = \{5, 7, 9, \ldots, 15\}$. This resulted in a grid with 216 different possible configurations for the number of basis. We iterated each row of the grid in a for loop to train 216 different GAM models, following the roll-forward validation strategy

introduced in Section 2.2. To obtain the best models, the accuracy metrics were measured with the predictions of each model against the test sets. The top 5 models which minimised the *RMSE* metrics are shown in Table 2.6. The models shown in the table are expressed using the formula notation used by the `gam` function.

	Model	RMSE	MAE	MAPE
1	$n \sim s(lag1, k = 23) + s(day, k = 27) + s(t, k = 11)$	4051	3563	13.6
2	$n \sim s(lag1, k = 23) + s(day, k = 24) + s(t, k = 11)$	4202	3717	14.2
3	$n \sim s(lag1, k = 23) + s(day, k = 21) + s(t, k = 11)$	4211	3679	14.1
4	$n \sim s(lag1, k = 20) + s(day, k = 24) + s(t, k = 13)$	4217	3725	13.8
5	$n \sim s(lag1, k = 20) + s(day, k = 21) + s(t, k = 13)$	4273	3777	14.1

Table 2.6: Top 5 models obtained after training 216 different GAM models, each with different number of basis for the smoothing functions. The models are expressed using the formula notation used by the `gam` function of the `mgcv` package.

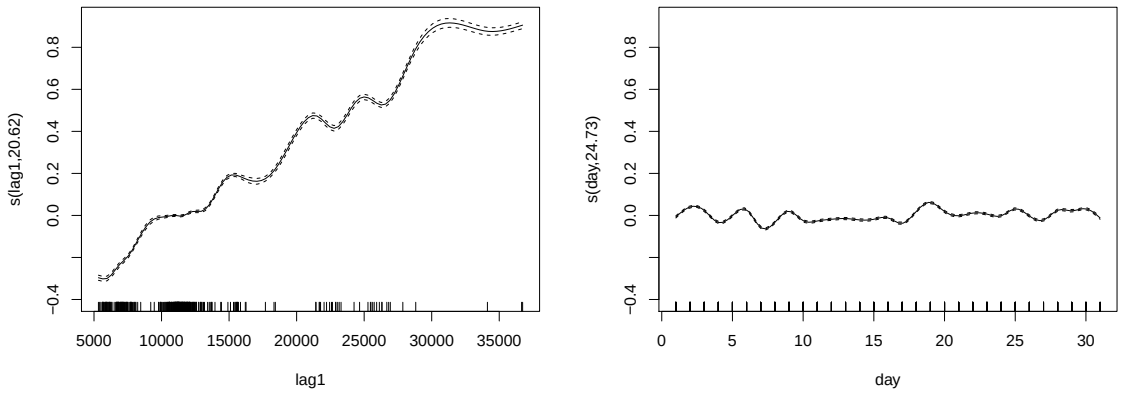
We notice the best results are obtained when the number of basis of the variables *lag1* and *day* are above 20. For the representation of the variable t, it was found that 11 basis were enough.

The model for which the *RMSE* is smallest, also minimises the *MAE* and the *MAPE* metrics. The optimal number of basis found wre $Nbasis_{lag1} = 23$, $Nbasis_{day} = 27$, and $Nbasis_t = 11$. Having the optimal values for number of basis for each smoothing function, we adjust the GAM model 2.3, and obtain the results shown in Table 2.7. The results show that all variables are statistically significant; the intercept coefficient, B_0, has a value of 9.3, which is the predicted value given all the other variables are zero. The `edf` column shows the effective degrees of freedom. This value represents the complexity of the smooth, where an `edf` close to 1 would be equivalent to a straight line. Figure 2.8 shows the plots of the 3 smooth functions of the model. They show us their effect in the prediction of the output, assuming the other variables remain fixed. Note that the functions are centered around zero. The effects of the smooth term for the variable $lag1$ is shown in Figure 2.8a, where we see its contribution to the prediction is proportional to its values. The effect of the variables *day* and t, which was introduced to capture the trend and seasonality of the series, can be seen in Figures 2.8b and 2.8c, respectively.

Parametric coefficients				
	Estimate	Std. error	z value	Pr(> |z|)
B_0	9.3000872	0.0005345	17398	<2e-16 ***

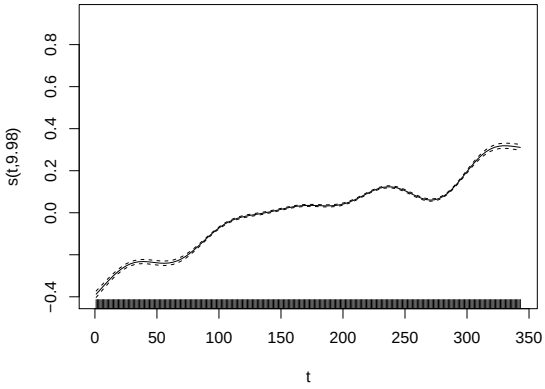
Approximate significance of smooth terms				
	edf	Ref.df	Chi.sq	p-value
$f_1(lag1)$	20.616	22	20588	<2e-16 ***
$f_2(day)$	24.735	26	2602	<2e-16 ***
$f_3(t)$	9.978	10	4998	<2e-16 ***

Table 2.7: Results of fitting the GAM model as defined by Equation 2.3.



(a)

(b)



(c)

Figure 2.8: Figure showing the fitted smooth functions.

We calculated the predictions for the days between 25 March 2019, and 31 March 2019, as shown in Table 2.8. In Figure 2.9 we see the graphical depiction of the predicted values, as well as the part of the series used in the test set. We see the predictions are very close to the values of the test set. The model was able to capture the trend of the data, and generalised well to the unseen data. These results are reflected by the accuracy measures that were calculated using the test data, shown in Table 2.9. We see $MAPE$ is just 1.66%, a very low value which shows the predictions were very close to the real observations; the value of the test $RMSE$ was 698.273, very similar to the value of MAE, with 620.571.

	Prediction	True value	Date
1	38189	37631	2019-03-25
2	37852	37075	2019-03-26
3	36840	35966	2019-03-27
4	37793	36672	2019-03-28
5	38379	38048	2019-03-29
6	39182	39783	2019-03-30
7	38093	38011	2019-03-31

Table 2.8: Table containing the predictions obtained for the last 7 days of the series, the and its true values, used in the test set.

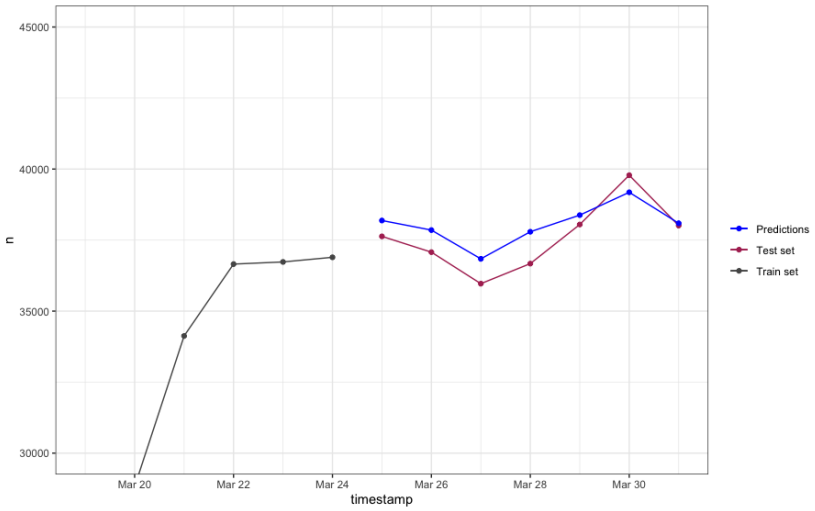


Figure 2.9: Graph of predicted values for the next 7 days, using the GAM algorithm. The training set is shown in dark gray, the test set is shown in dark red, and the predictions are in blue.

RMSE	MAE	MAPE
698.273	620.571	1.666

Table 2.9: Accuracy metrics calculated with the predictions compared with the data from the test set.

2.2.3 Remarks about the results in this chapter

Comparing the results of both models, we conclude the fitted model with the GAM algorithm gave the best performance. The GAM model was able to obtain a test $MAPE$ of 1.666%, while the the $MAPE$ of random forest was 5.48%. The other two metrics, $RMSE$ and MAE, were also considerably smaller for the GAM model. The inclusion of the smooth terms for the variables $lag1$, day and t were enough to get good predictions for the number of attacks for any given day. We should also mention that in the cross validation stage, the random forest algorithm had better performance with an $RMSE = 3423.610$, $MAE = 2937.089$ and $MAPE = 11.12$, while accuracy metrics for the GAM algorithm were $RMSE = 4051$, $MAE = 3563$ and $MAPE = 13.6$, suggesting the random forest algorithm could be overfitting the training data.

We are going to conclude this chapter by mentioning some areas of opportunity and ideas that could further improve the performance of the forecasting analyses.

- Initially, we thought of including variable $t = 1, 2, \ldots, T$, in the random forest analysis. In the end, it was not included, upon realising the trained model will be making predictions using values of t that were not included in the training.
- During the feature engineering step prior to training the decision trees, all the variables that were added (see Table 2.2) were encoded as numeric, even though some variables could have been encoded as categorical, such as *month*. This was done because when we tried to calculate the predictions while using the roll-forward validation approach, the code would throw an error if the data used to calculate the predictions had level of a categorical variable that was not included during the training of the model.
- When analysing the results of the random forest we came to the realisation that perhaps, because the series had a positive trend, the random forest algorithm was never going to give appropriate predictions if the trend kept going up, as it would always underestimate the number of attacks. We have to keep in mind that random forests for regression, calculates the predictions based on the average values of past observations. The random forest algorithm is not suitable for this type of data.
- Given that it is very likely that the pattern of number of attacks by day changes with time, it could be desirable to give more importance to more recent dates, as old data can become irrelevant. One could think of using an *exponential smoothing* method to the data, as they are weighted averages of past observations, where weights decay as the observations get older.
- Additional work could be done in order to automatically select the variables used for the GAM algorithm. A more rigorous approach could be done, for example, by doing hypothesis testing of various models (see Wood, 2006, chap. 2).

Bibliography

Aggarwal, C. C. and Reddy, C. K., editors (2014). *Data Clustering: Algorithms and Applications*. CRC Press.

Allaoui, M., Kherfi, M. L., and Cheriet, A. (2020). Considerably improving clustering algorithms using umap dimensionality reduction technique: A comparative study. In El Moataz, A., Mammass, D., Mansouri, A., and Nouboud, F., editors, *Image and Signal Processing*, pages 317–325, Cham. Springer International Publishing.

Arthur, D. and Vassilvitskii, S. (2007). k-means++: the advantages of careful seeding. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Arvai, K. (2020). Kneedle: Knee-point detection in python. https://github.com/arvkevi/kneed. Last checked on June 15, 2020.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth and Brooks.

Celebi, M. E., Kingravi, H. A., and Vela, P. A. (2013). A comparative study of efficient initialization methods for the k-means clustering algorithm. *ArXiv*, abs/1209.1960.

Davies, D. L. and Bouldin, D. W. (1979). A cluster separation measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):224–227.

Dominici, F., McDermott, A., Zeger, S. L., and Samet, J. M. (2002). On the Use of Generalized Additive Models in Time-Series Studies of Air Pollution and Health. *American Journal of Epidemiology*, 156(3):193–203.

Dua, S. and Du, X. (2011). *Data Mining and Machine Learning in Cybersecurity*. Auerbach Publications, USA, 1st edition.

Dunn, J. (1973). A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters.

Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press.

Fix, E. and Hodges, J. L. (1989). Discriminatory analysis. nonparametric discrimination: Consistency properties. *International Statistical Review/Revue Internationale de Statistique*, 57(3):238–247.

Forgy, E. (1965). Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics*, 21:768–780.

Hartigan, J. and Wong, M. (1979). Algorithm AS 136: A K-means clustering algorithm. *Applied Statistics*, pages 100–108.

Hastie, T. and Tibshirani, R. (1986). Generalized additive models. *Statist. Sci.*, 1(3):297–310.

Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA.

Huang, Z. (1997a). Clustering large data sets with mixed numeric and categorical values. In *In The First Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 21–34.

Huang, Z. (1997b). A fast clustering algorithm to cluster very large categorical data sets in data mining. In *In Research Issues on Data Mining and Knowledge Discovery*, pages 1–8.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning: with Applications in R*. Springer.

Kaufman, L. and Rousseeuw, P. J. (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley.

MacQueen, J. B. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297.

McInnes, L., Healy, J., and Melville, J. (2018). Umap: Uniform manifold approximation and projection for dimension reduction.

Milligan, G. W. (1981). A monte carlo study of thirty internal criterion measures for cluster analysis. *Psychometrika*, 46(2):187–199.

Milligan, G. W. and Cooper, M. C. (1988). A study of standardization of variables in cluster analysis. *Journal of Classification*, 5(2):181–204.

Nelder, J. A. and Wedderburn, R. W. M. (1972). Generalized linear models. *Journal of the Royal Statistical Society. Series A (General)*, 135(3):370–384.

Oosterhof, M. (2018). Cowrie documentation. https://cowrie.readthedocs.io/en/latest/index.html. Last checked on June 15, 2020.

Raschka, S. (2015). *Python Machine Learning*. Packt Publishing - ebooks Account.

Sander, J., Ester, M., Kriegel, H.-P., and Xu, X. (1998). Density-based clustering in spatial databases: The algorithm gdbscan and its applications. *Data Mining and Knowledge Discovery*, 2(2):169–194.

Satopaa, V., Albrecht, J., Irwin, D., and Raghavan, B. (2011). Finding a kneedle in a haystack: Detecting knee points in system behavior. pages 166 – 171.

Schubert, E., Sander, J., Ester, M., Kriegel, H. P., and Xu, X. (2017). Dbscan revisited: Why and how you should (still) use dbscan. *ACM Trans. Database Syst.*, 42(3).

Stankowicz, J. and Kuzdeba, S. (2021). Unsupervised emitter clustering through deep manifold learning. In *2021 IEEE 11th Annual Computing and Communication Workshop and Conference, CCWC 2021*, pages 732–737.

Tobías, A. and Saez, M. (2004). Time-series regression models to study the short-term effects of environmental factors on health. Working Papers of the Department of Economics, University of Girona 11, Department of Economics, University of Girona.

Wood, S. (2006). *Generalized Additive Models: An Introduction With R*, volume 66. Chapman and Hall.