Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Simon Rødsbakken Røe

Identifying Malicious Python Packages

Using Static Indicators and Machine Learning

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
Co-supervisor: Felix Leder
June 2023

Simon Rødsbakken Røe

Identifying Malicious Python Packages

Using Static Indicators and Machine Learning

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
Co-supervisor: Felix Leder
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Abstract

The sharing culture, especially Open-Source, has become important with the increased use of technology and the internet. In programming, languages like Python rely on the open-source community to create libraries for people to use. This allows anyone to implement complex application functionality simply by adding a package.

However, the danger of letting anyone publish such packages is the possibility of evil actors trying to exploit the users by uploading malicious packages, which can cause significant damage if they can sneak into popular projects.

The thesis aims to increase the domain knowledge of the characteristics commonly found in malicious packages in the Python Package Index (PyPi) and to see how common machine learning models perform on this data. We have used static indicators and metadata of a dataset consisting of 382,712 benign and 7,639 malicious Python packages. We have used the feature ranking method Information-Gain and common machine learning models from the Python library Scikitlearn to find the most valuable features and identify the expected performance from the models.

From the experiments conducted, did we find the Neural Network Perceptron model to perform the best with the default options among those we tested with an F1-score of 92% against the verification dataset. We also found the most common features among the malicious packages to be two commonly used libraries, "requests" and "setuptools". From the results of testing the models, we found that it can identify most of the small malicious samples while the average size of those misclassified was much higher. It needs to be looked closer at improving the detection of larger and potentially more sophisticated packages.

We conclude that static indicators and machine learning could be good for detecting malicious packages. However, more research into optimization is needed, as well as more profound knowledge of what combination of indicators is typically malicious. We have contributed with extended knowledge on the topic and on how some models perform on static indicators on a larger dataset of packages. We have also provided recommendations for what could be focused on further in research on the topic.

Sammendrag

Delingskulturen som finnes på internett og spesielt knyttet til åpen kildekode har vært viktig ved den økende interessen for teknologi. Innenfor programmering og språk slik som Python lener seg på gruppen som produserer og deler sine prosjekter med andre i form av programmer og funksjonalitet.

Men, faren ved å stole på andre gjør det mulig for ondsinnede aktører å utnytte vanlige brukere ved å lokke med god funksjonalitet. Ved å legge med skadelig kode i disse pakkene som deles kan føre til store konsekvenser hvis de klarer å snike seg med i større programvareprosjekter.

Denne oppgaven har som mål å øke kunnskapsnivået om attributter som er vanlig å finne på slike ondsinnede pakker i Python sitt pakkebibliotek PyPi samt undersøke hva slags prestasjon kan forventes av de vanligste maskinlærings modellene. Vi har gjort eksperimenter på metadata og statiske indikatorer hentet fra ett datasett som inneholder 382,712 legitime og 7,639 ondsinnede Python pakker. Vi har benyttet oss av InformationGain algoritmen for å rangere indikatorene og modeller hentet fra maskinlærings-biblioteket Scikitlearn for å finne de beste indikatorene samt å identifisere forventet presisjon innen klassifisering.

Fra eksperimentene fant vi at Neural Network modellen Multilayer Perceptron presterte best med de originale parameterne. Det høyeste F1 resultatet fra tester mot datasettet for verifisering viste 92%. Vi fant også at de vanligste indikatorene blant ondsinnede pakker var de to godt brukte bibliotekene "requests" og "setuptools". Resultatene fra testing av modeller fremstår det som vi klarer å detektere mange av de små ondsinnede pakkene, siden gjennomsnittlig størrelse var mye høyere blant de feilklassifiserte. Det må derfor trolig ses nærmere på hvordan forbedre deteksjon av større og potensielt mer sofistikerte pakker.

Vi konkluderer med at statiske indikatorer og maskinlæring kan være en god kombinasjon for å detektere ondsinnede pakker. Men det trengs mer forskning på hvordan optimalisere hver modell samt mer kunnskap om hvilke kombinasjoner av indikatorer er oftest sett blant de ondsinnede pakkene. Vi har bidratt med mer kunnskap, blant annet om hvordan ulike modeller klarer å klassifisere ondsinnede pakker basert på metadata og statiske indikatorer. Vi har også kommet med forslag til hva som kan fokuseres på videre innen dette området.

Acknowledgements

I would like to express my deepest gratitude and appreciation to all those who have supported and guided me throughout the completion of this master's degree

First and foremost, I am immensely grateful for all the help and guidance I have received from my supervisor, Dr. Geir Olav Dyrkolbotn, CCIS, NTNU and co-supervisor, Dr. Felix Leder, Crosspoint Labs. This thesis would not be possible without their patience, knowledge, and insightful feedback.
I am also extremely grateful for the opportunity I was given by ReversingLabs to conduct this thesis. Without their contributions, this would not be possible.

I especially want to thank my girlfriend, Anine, for always being supportive and helping me through tough times. I sincerely appreciate my family and friends for their constant encouragement and understanding. Their love, belief in my abilities, and unwavering support have inspired and motivated me during this journey.

Contents

Figures

Tables

Chapter 1

Introduction

This chapter will introduce the thesis by presenting some of the challenges users of open-source software face today, before describing the motivation for conducting such a thesis and then what contributions we aim to provide.

1.1 Background

Open-source or free software has been along for a long time and developers can today achieve a lot and save themselves a good amount of work by utilizing free libraries. As most programming languages have packages of code that can be implemented to easily solve certain tasks. One package repository for the well-known language Python is called Python Package Index(PyPi) and counted on an ordinary day in February 2023 4,489,234,921 downloads across all packages available[1]. This shows to what extent the repository of only this language is used weekly. Since these packages are open-source, in theory, everyone can contribute and that opens the possibility for malicious actors. If a malicious package ends up being used in a project widely used, that one project could risk being affected because of the one bad package being used as a dependency. This is called supply chain attacks and is a field that has gained a lot of interest in recent years after being the reason for several significant security incidents. As more come to realize the importance of securing the supply chain, more interest has been in developing tools and methods to verify what is put into the supply chain and not only focus on the end product. By analyzing the data about known bad packages and comparing it against legitimate packages, could we assist in the development of indicators to use in combating fake and malicious packages. According to the paper by Vu *et al.* [2] the results of comparing different detection approaches that there was a high number of false positives present. Showing the need for further work towards making it easier to prioritize and further analyze individual packages.

1.2 Motivation

Open-source projects and libraries are a large and essential part of the software development process, both for people coding as a hobby and for professionals working for big companies. Part of the motivation for doing this project is the ability to contribute back to the open-source community, which is mainly based on voluntary effort, and to aid in the security itself by improving on the knowledge about malicious packages located in these repositories. Part of the motivation is also based on the work done by ReversingLabs, experts, and other companies regarding supply chain attacks and research into open-source security. Improving the security and methods to detect malicious packages will not only benefit the users but also aid the security companies as they hopefully can improve in detecting malicious packages with fewer resources or gain more intelligence on who is behind the campaigns. As reported by several companies, have there already been detected multiple new malicious packages in PyPi, early January where there reports of multiple evil packages circulating in PyPi[3]. As this issue not only appears in new and low populated packages were also in the large and well-used Machine Learning package "PyTorch" compromised late December last year[4] which shows us to be cautious whenever using existing libraries. Some of the initial motivation for conducting this project is also based on whether it is possible to identify malicious packages only based on metadata and static indicators, which will be used in this thesis. It will also be interesting to see whether some indicators tend to appear more often among the malicious samples or not.

1.3 Expected impact

After finishing this thesis, the goal is to provide more insight into what attributes are commonly found among especially malicious packages in PyPi. The expected impact of this thesis is firstly to provide more insight into what type of features are commonly found in malicious packages by looking into the static indicators in the data. Secondly, the thesis will look into what performance can be expected on this data for some of the common machine learning models that are easy for users to implement using existing libraries. More, the thesis aims to contribute towards improving the general knowledge about malicious packages and give some directions to what should be focused on in further research into machine learning to aid in identifying malicious packages.

1.3.1 Research questions

- *RQ1* What indicator or combination of indicators are most commonly found in malicious packages?
- *RQ2* What features are better at determining whether a package is malicious?

- *RQ3* What performance can be expected from machine learning models when applied on static indicators found in malicious packages?

1.4 Overview

After introducing the reader to the background and the main focus of the thesis, will we further present what has been done, how, and what results we have achieved. The structure of the thesis is made so the reader first will go through some background information regarding machine learning and what a package is before presenting related work. Afterward, will there come a chapter describing the methods used and give the reason for why certain decisions have been taken. Lastly, the results will be presented part by part as shown in the methodology, and a discussion of the results will come at the end accompanied by a section of feature work.

Chapter 2

Background

The next chapter is split into two parts to increase readability by separating the different aspects. The first introduces and focuses on the theoretical foundation upon which the thesis is built and will help in understanding the methodology and the concepts discussed in the thesis. For the second part, previous research and related work will be described to give an insight into what has already been conducted in this research field and to give reasons for why this thesis should be conducted.

2.1 Theory

This next section will describe the general theory about the technologies used during the thesis. One of the main aspects of this thesis is the programming language Python and its package manager PyPi which stores all additional libraries. It will briefly describe what it is and how it works before we will go through malware analysis and machine learning which is also important parts of the thesis.

2.1.1 Python

Python is one of several programming languages that have become highly popular and is seen used across many areas, first released in 1991[5]. Being a high-level language, the syntax of Python is straightforward, making it easy to learn and write, besides maintaining high readability for the users. Python is an interpreted language which means the code is compiled while being run instead of the need to be fully compiled before it can be used. It can also easily be extended with functions written in Python or the C language to make fast and effective additional functionality. This makes the language powerful and highly customizable, one reason for its popularity and widespread use. Python is open-source and owned by "The Python Software Foundation", making it available for everyone to use and contribute. A major part of the language is the "Python Package Index" (PyPi) which hosts all packages available by the official Python contributors and all other third-party modules contributed to by the open-source community.

2.1.2 Packages and package managers

As mentioned, PyPi is the official package manager for Python. It consists of most packages for implementing additional functionality, allowing developers to create new applications quickly. A package or library is code encapsulated and ready for use. It is either written by the official Python contributors or by the open-source community. The reusable code can be in the form of functions, small applications, or similar that allows users to directly implement functionality that is able to handle most tasks, such as utilizing machine learning models or saving data to a special format that the existing packages can enable. Within these packages are there usually one or multiple files present, which contain the code and additional content for it to work correctly. When the package is uploaded to PyPi, it is listed together with metadata that describes attributes such as the file size, dependencies, descriptions, and author contact information.

To put a number on how much code is available in PyPi, does the official page mention the need for 14,7TB free space to mirror the full repository with all packages and their multiple releases[6]. While another statistics page for PyPi, "pypistats.org"[1] did record 799,225,593 downloads on an arbitrary day, while the last month showed 20,643,058,415 downloads from all packages in the PyPi repository. Both the large size and the number of downloads show to some extent, the size of the Python ecosystem and how widely it is used.

2.1.3 Malware analysis

Since the internet was created, malware has been developed and distributed intentionally and accidentally. As technology has evolved alongside the increased use of technical devices, the number of malware and level of sophistication increased. Malware is a shortened name for malicious software and is used to describe code designed to cause harm or lead to infections. Some of the most common malware types are mentioned in the well-known malware analysis book "Practical malware analysis" Sikorski [7] and are the following: viruses, Trojan horses, worms, rootkits, scare-ware, and spyware. One infamous addition to the list is ransomware which has been widely used in later years to spread havoc in several industries. In order to handle encounters with different malware, knowledge about how it works and what is done is necessary, both for preventive measures and for cleaning up all bits and pieces afterward. Malware analysis is the term used for this activity, and one description is "Malware analysis is the art of dissecting malware to understand how it works, how to identify it, and how to defeat or eliminate it."(Sikorski [7] p. xxviii)

As one of the reasons for conducting this thesis is based on the occurrences of malware on the internet. This section is included to improve understanding of where the data comes from and why it could be helpful in such a use case. Malware analysis is the umbrella term used for describing how malware is investigated and understood by researchers. Within malware analysis, there are mainly two common methods to inspect a malware sample, either "static analysis" or "dynamic analysis", and these concepts will be touched upon in the following sections.

Compiled vs. interpreted code

In order to run programs on a machine, the code itself needs to be compiled or interpreted before it is able to run. When talking about compiled programs, some well-known languages are C and C++. These two are examples of languages that converts the code into machine code which is hard to read for users but consists of simple commands for the machine. In order to run programs written in either of these, a compiler needs to be used to compile the program and make it runnable. The text representation is lost in the process, and the newly created program only contains the machine code. However, it runs on most computers but may need a specific library or software to work. Different from languages that need a compiler are interpreted languages such as Python, which needs an interpreter for the code to work. These languages are known under the term "higher-level" languages and are interpreted or compiled during run time and do need a specific interpreter to run. The code does not lose its text representation and is just as easy or hard to read before it is run as after, making it easier to inspect than compiled code. The two methods for malware analysis will be described below but can just as well be used on interpreted code, where the code can be viewed directly in a text editor.

Static

The first category of methods, known as Static analysis, comes in two parts, basic and advanced analysis. One of the first steps in malware analysis is the basic static analysis which involves techniques and tools for examining the suspect file without running it but looking at its properties. This step indicates whether a file is malicious or not and could provide information on some of its functionality by listing strings and modules, it utilizes. In this thesis the data consists of indicators which are found by utilizing static analysis tools to extract these characteristics of the samples. However, dealing with more advanced samples, these techniques might still be ineffective as the writer most likely has utilized techniques to cover their tracks.[7]

For the more advanced part in this category, advanced static analysis requires a higher degree of knowledge, at least when handling compiled samples. The method is based on reverse-engineering the specific malware sample by using a disassembler which gives the analyst a list of instructions on what the sample is programmed to do.

Dynamic

Dynamic analysis means running the malware in a secure and isolated sandbox. Same as for static analysis, the dynamic method is divided into basic and advanced, where basic is about running and observing the consequences of the sample. Extra programs are often used to record the changes conducted by the sample to see what activity is conducted whether it is to store something to the disk, make network connections, or other harmful activities. The advanced option is to run the sample using a debugger to follow the code while it executes manually. The benefit of running the code in question is the ability to look at what happens when the program is executed and inspect the changes made step by step. This is normally the case, but more advanced samples can contain certain techniques and capabilities that hide their malicious intent if it detects it's running in a sandbox or similar and demands the use of advanced methodologies to manage to identify the actual activity.

2.1.4 Machine learning

Machine learning (ML) has been a hot topic in computer science in recent years, and many industries have picked up on this technology. It is commonly used on large amounts of data to detect patterns that are hard for humans to understand and recognize. The need for such tools has increased because the amount of data that is generated makes it impossible for humans to manually inspect and find patterns. There exist many different models that vary from simple to advanced and complicated methods and algorithms. ML models are mainly based on math and statistics, and the decision to predict in one direction is based on the underlying algorithms applied to the training set. This section will go through some common machine learning (ML) models and the process for conducting such projects.

Process

When implementing a machine learning project is there several common steps to go through. I will briefly describe the main aspects of the process and what steps are taken. However, it will not go in-depth into all the small steps which are used but give a broader overview of how such a project is structured.

The first step in an ML process is to gather and preprocess the data. After the data has been acquired is it necessary to clean and structure the data so the models later on can interpret it. One common way to store all the data entries is to convert them to a Comma Separated Values (CSV) file where all data has the same structure. During this process are the data also normalized and converted to an appropriate format. This means that features with numbers of different sizes are converted to the same scale for the model to later easier interpret and not only favor the highest numbers. Categorical variables are also handled, one common way is to convert the categories into a binary list and include them as features for the samples.

After the data have been structured is it time to get familiar with, and identify the best features to use. For getting familiar with the data is it common to use visualizing and statistical methods to get an understanding of what type of data is used and to understand what features might belong together. During this process is also the composition of samples is considered, whether the datasets are balanced and contain a similar number of samples representing each of the categories to divide into. If the amount of samples in each class differs is the dataset imbalanced, and we usually use the term minority class for the one with lower representation and the major class for the one over-represented. Then to identify the most prominent features does it exist several methods and algorithms. Within the categories of filters, wrappers, and embedded methods are there several algorithms to choose from to rank the features. One method which will be described more in Section 3.3.1 is Information Gain (IG) which belongs to the filter method. It is computationally inexpensive compared to other methods, which makes it fast and easy to implement. Then, after a set of features has been identified are the data split into three parts, which are training, testing, and validation set which will be utilized in the next part.

After the data have been prepared, is it time to decide on a model to use and train it. There are several categories of ML models, and some of them will be described in the following sections. What model to choose depends on the type of problem to solve and in some cases, it could be beneficial to test multiple up against each other to see what performs best. After the model is selected, can it be fitted on the training data so it can learn patterns based on the algorithms used in the model.

When the model is fitted with the training data is it ready to be evaluated on the test data. Based on what the model learned when being fitted does it now try to predict what is found in the test data. The results from this are usually a confusion matrix which consists of how many of the samples were correctly classified and how many were wrong for the classes used. To evaluate the performance of the model several metrics can be used, some of which will be described in Section 3.1.2.

Based on how the model performed in the previous stage it can now be optimized by testing various parameters and options that are available for the model. The last step is then to retest the final model on the verification data which were saved throughout the process in order to verify the results on a new dataset. These final results can then give an indication of how well the model performs, at least for the data which were used.

Supervised vs unsupervised

Machine learning models are mainly classified into two categories called supervised and unsupervised learning. On a high level, the supervised learning models are trained by being fed data where the correct label is provided. The user guides these models to understand what features and traits are common for each category it can be divided into. When using unsupervised models, they are provided training data that do not contain a label or correct solution. These types of models have to compare and interpret the features describing the characteristics of each sample and divide them into probable groups. Having its differences, each type is specified to handle certain types of problems and use cases. On the other hand, it is common for both to classify samples into different groups and categories or do regression, which is used to describe and predict statistical changes.[8]

Models

The first category of model architectures is distance based. These have in common that they base the classification on the distance measured between features that it is trained on. Using math and statistical algorithms, they can classify new samples by computing a score of all the features for each sample and predict the class with the closest score. The first model is called K-nearest Neighbours and uses the training set to divide into a set number of neighbors which the user decides. The training data is divided into the number of neighbors based on their calculated similarities. New samples are predicted to be a specific category based on which neighbors are closest. The second algorithm included is Support Vector Machine(SVM), also mentioned as SVC as that is the package's name in Scikitlearn. This model draws hyperplanes in multidimensional space to separate the plotted groups and makes predictions based on what side of the lines the new sample is plotted.

Multilayer Perceptron(MLP) is a fundamental type in the ML subfield called "Deep learning" where Artificial Neural Networks (ANN) are used. The name comes from the way the model is built by multiple layers of nodes also called "perceptrons". The network of nodes consists of one input layer, one or more hidden layers before the last output layer. Each node does a computation on the input they receive with an activation function which produces output that is sent to the next layer. The complexity of the model and the network of nodes increases with the number of layers and perceptrons, but deeper models are able to learn intricate relationships between the data and offer a great deal of flexibility for the users making it highly popular. Being a complex model understanding how it works and why it takes certain decisions is harder to understand compared to others such as models based on tree structures which are possible to debug.

The second architecture category included in the thesis is rule-based algorithms and, in this case, a few variations based on tree algorithms. The first is Decision Trees which create a binary tree structure where the dataset is split on specific parameters deciding whether to go down one path of the tree or another. Each new step splits the remaining data into partitions until the leaf nodes or end decisions can categorize all training data. [8] Being a quite simple model it is easy to interpret as it is possible to view each decision that is taken and see the reason why each sample was categorized as it was. However, it will end up differently each time as it depends on the training data and can lead to different results with each build. The second model in this category that is included is the Random Forest which is based on using multiple decision trees where each individual uses a random subset of the training data. The full model can then use the statistics from all generated trees to build a better-performing model. The last model included in this category is the Gradient-boosted trees which is also a method based on decision trees where one and one are added at the time, correcting the mistakes of the previous one and therefore giving the name "boosting". As this model learns from the previous in the attempts to increase accuracy, it is vulnerable to overfitting.

The last category and final model to be added to the comparison is Naive Bayes which is a well-known model and the oldest among statistical methods. Naive Bayes makes assumptions as to what class each sample belongs to based on statistics of the different features present. Even though the simplistic approach of using probability, it has in some cases, shown to perform quite well on real-life problems such as spam classification[8].

Cross-validation

When implementing machine learning models, various methods are used to validate the results achieved during the training phase. When experimenting and training models the dataset is usually split into two parts where one is used for training and the second for testing. In this process, cross-validation can be included to verify whether the achieved results are representable for the whole dataset or if some irregularities appeared. This could be a poor selection of the type of samples used in testing the data, resulting in reduced performance as the model has not seen the data that only appear in the test set. The method chosen to be used in this thesis is K-fold. The K in K-fold is the number of parts the dataset will be split into. When using cross-validation, the dataset in use is split into K equal parts, and in turn, one part is being used as the testing set while the others form the training data for the ML model. For each fold, the same procedure will be conducted, and looking at the average of the predicted results can help the reader better determine how well the model performs on different compositions of the same dataset. Cross-validation is used to improve the performance of the ML algorithm. If you train on a single selection of the data, there is always a chance that the training data or the test data is biased. This bias can give you an artificially high or low accuracy. Repeating the random split of the data K-times will average out such biases[8].

2.2 Related work

Supply chain security is a field that has gained a lot of interest in recent years after being the reason for several significant security incidents. As more come to realize the importance of securing the supply chain, more interest has been in developing tools and methods to verify what is put into the supply chain and not only focus on the end product. This thesis will look at lower-level security by looking at how to identify malicious packages in the package repository, especially for Python. When looking closer at the interpreted languages that do not need to be compiled to run, Python, JavaScript, and Ruby are commonly looked at together and separately in research papers. This can be seen in articles looking at malicious packages and their package managers since all mentioned languages, use an open-source repository to provide their users with extra code libraries. Some papers include the package managers of all three, like the paper from Ohm *et al.* [9], while others focus on one language and package manager like [10]. An observation from the viewed literature is that it tends to be more research on the JavaScript package manager NPM than the two others mentioned. The related works described in the following chapter are divided into three pieces. The first section focuses on research on package managers and packages in general. The second part will consist of papers describing the threats and security of the package repositories. At the same time, the last piece focuses on what has been done to detect and prevent malicious packages.

2.2.1 Analyzing malicious packages

Bommarito and Bommarito [10] did an empirical summary of 178,592 packages and 1,745,744 releases from PyPi, looking at package metadata and source code. From their results, they state that PyPi had a growth rate of around 47% for actively maintained packages and 39% for new authors publishing and contributing to existing packages. Meaning the number of actively used packages rapidly increases simultaneously with the number of contributors and authors. From their statistics, they experienced a skewed distribution by looking at releases per package, releases per author, and imports per package. Where a small number of packages accounted for a large percentage of the statistics, which means some few packages are very actively maintained while most others are not. Another exciting finding after looking at a large portion of containers is that most are contributed to by individuals, not organizations or multiple people.

Ruohonen *et al.* [11] conducted a large-scale static analysis on Python packages focusing on security. The researchers took a snapshot of the PyPi, and the analysis was done on over 197 000 packages, discovering more than 749 000 security issues. Even relying only on static analysis, they found many issues, and around 46% of the packages contained at least one security vulnerability. The most common type of issues found were related to exception handling and code injections.

2.2.2 Threats and security in package repositories

One of the first comprehensive surveys of potential risks in Open source libraries also called registries is by Kaplan and Qian [12]. As they look into the top registries, they point out multiple weaknesses, one being that contributors only need to sign up by email before publishing packages. This makes it easy for attackers to upload packages without revealing their identity. Some of the registries do not have any control mechanisms in place that can detect malicious code. They rely on user reports and admins' inspections to combat malicious packages. This can also lead to some packages having a long lifespan before removal. When they wrote their research, they also understood that only ten people had access to remove packages, and they were responsible for around 40k developers each. As the number of people and resources available for working on securing the repositories are low and few compared to the massive amounts of packages have made companies and researchers put their focus on detecting malicious samples in the repositories. This leads continuously to new discoveries and further improved knowledge on techniques used by attackers[13].

When discussing some of the common attack vectors, typosquatting, and combosquatting are mentioned. These are commonly found in all of the Open Source(OS) registries and attempt to trick the developers into installing and using the wrong package. This is done by uploading packages with similar names as those that already exist and waiting for users to install the wrong package. The paper by Ohm *et al.* [9] looked closer at a dataset of malicious packages and identified that 61% of the malicious packages they analyzed mimicked existing packages by name. The average Levhenstein distance was about 2.3, where the number refers to how many edits are done between the two names. Even though some switch out a letter or two in an attempt to disguise themselves, others have also been seen to switch the order of words, like "kafka-python" for "python-kafka," which can fool a simple test like Levenshtein. Ladisa *et al.* [14] have also looked into the taxonomy of attacks found in OS software (OSS). Other common attack vectors for actors targeting OSS are also described. Most normal is attempting to compromise highly valued accounts or to become maintainers in order to inject malicious code into already popular packages. This has been found during different investigations and does seem to be a big problem, at least when the number of preventive measures is relatively low.

Some challenges mentioned by Kaplan and Qian [12] are based on heavy code reuse and trivial packages. Trivial packages only have a few lines of code and implement simple functionality are often used by developers to improve efficiency and reduce the amount of code needed. Using many small packages can expand the attack surface for a threat actor and make it easier to target. Code reuse is also an issue, especially in NPM, as they mention that packages rely on several other packages and make the tree of dependencies long and hard to maintain. This could also make it easier for an attacker to sneak in a malicious package that, in the worst case, can impact a large crowd using a single dependency which could potentially spread further. As found by Duan *et al.* [15], several malicious packages were downloaded over 100 000 times and show the magnitude it could potentially have. The danger malicious packages introduce is that when they manage to inject themselves inside a larger chain of dependencies or into more popular ones, it allows them to be part of larger software packages widely used. They can then be hard to detect as only one small package is hidden among the others, which can cause a compromise. Another paper by Zerouali *et al.* [16] found the average time it took to fix half of all vulnerabilities was 55 months in NPM and 94 months in RubyGems, showing how long time attackers potentially have to exploit the weaknesses or wait and let it spread to new project or apps.

2.2.3 Detecting malicious packages

After looking into threats and the general security of the top OS repositories, it is time to view what has been done regarding how malicious packages can be detected, which techniques are implemented, and what technology could be beneficial.

A wide collection of tools have been created for detecting malicious packages, and Vu *et al.* [2] compares several of them in a benchmark to identify their strengths and weaknesses as well as to understand how well they perform. The different automatic detection programs are tested on a dataset with 168 malicious, 1430 popular benign, and another 986 randomly selected benign packages. One of the things they identified was the goal of the maintainers, which is to mitigate most of the low-effort attacks with a minimum amount of effort. For the workload to be manageable, they cannot aim to remove all malicious packages, and any tools they implement need to have close to zero false positives. Otherwise, they will not be able to handle it. The result from testing the different automated tools is between 50% and 66% percent detected rate. A downside of the results they discover is the false positive rates are high and sometimes even higher than the rate of true positives, making them hard for the repository maintainers to utilize.

Besides automated detection programs, applications like "spellbound" are presented by Taylor *et al.* [17] and focus on detecting and defending against typosquatting attacks. SpellBound is designed to be an easy solution for users as it is integrated on the user end into the package manager. It will defend the users by checking the requested package by name with similarly named packages. It also has the ability to do specific transformations to detect a larger number of possible name variations. Suppose it detects other more popular packages with the same or similar name. Will the user be alerted and have the ability to abort the installation to investigate further and verify that the correct package is installed.

Another contribution towards securing the supply chain is made from Vu [18], which presents the tool "PY2SRC". The purpose of this tool was to ease the job for the developers when choosing which libraries to implement in their projects by automatically identifying and providing the sources of the specific PyPi packages. From their results, "PY2SRC" outperform the other information sources they tested, and it provides reliable attribution metrics for verifying if the link to the code corresponds to what is expected.

Not only have tools been created for the purpose of combating malware, but in general, are researchers trying to prevent attacks from all platforms and surfaces. Bringing in new technology is nothing new, and several attempts have been conducted trying to utilize ML in detecting malware. One of the more common places malware is found is in the Windows ecosystem. Surveys for using ML have been made by both Ucci *et al.* [19] and Shalaginov *et al.* [20]. The last also provides a tutorial for how ML models could be implemented and used to detect compiled executables. Freely available ML tools are listed, and the full methodology to use ML is provided together with well-known algorithms that can be used for feature extraction and selection and describing some models that could be used for classification. However, both were mostly directed toward handling Windows executable files. Milje [21] attempts to develop a new method for analyzing Python packages and detecting malware. In that research, static analysis features were extracted from the samples by using the Abstract Syntax Tree(AST) to show function calls, imported modules, and variables. Rules and weights were also used for deciding the degree of maliciousness of each sample tested and by combining a certain amount of indicators or features, it is deemed malicious.

Chapter 3

Methodology

The thesis is a quantitative research project that is based on a dataset of malicious and benign packages collected by ReversingLabs. Based on the timeframe, the focus will be on broader experiments to answer the research questions(RQ) and identify what could be done in future research into the topics.

The thesis aims to identify what features most often appear in malicious Python packages compared to legitimate ones. It is also to improve the domain knowledge on this topic and look at what features they contain to answer RQ1 and RQ2. To answer RQ3 about how Machine Learning(ML) models perform, will several models be tested to see how well they can predict what is malicious or not only using static indicators. This will be better described later in the chapter. Results from testing the different models will be a sub-goal of improving the knowledge about what characteristics are found in the packages. As it will show how well different models can handle this type of data and specifically the dataset used in the project. The results will also be a foundation for further research into choosing and optimizing ML models to solve similar challenges. For gathering statistics and identifying the features better suited for separating malicious and legitimate packages feature ranking algorithms will be utilized together with some of the ML models which have the ability to describe what conditions are used for categorizing samples. Before describing what models and methods to use the origin and structure of the dataset used will be discussed as well as tools and techniques.

3.1 Technology

Much of the thesis utilizes technology, especially ML, to see how different models perform. And when using ML, do we also need metrics to evaluate how well they can perform. This section will describe what type of ML models to use and how each model will be trained and tested. It will then describe what metrics will be used to present and compare the results achieved.

3.1.1 Machine learning

One of the main technologies that will be utilized during this thesis is ML. Some applications allow for easy implementation of ML methods, such as Weka and other free alternatives, as mentioned by Shalaginov *et al.* [20]. However, to easier handle and work on the results and to specify our needs, the choice fell on using Python with a machine learning library. Several libraries are available for this type of work, some being TensorFlow[22], Pytorch[23], and SciKitLearn[24]. The choice fell on SciKitLearn as it contains most of the well-known algorithms and other ML functionality, as well as being well documented with multiple tutorials and explanations available online. Using Python instead of other tools makes it possible to design the experiments by writing them in code, making it easy to repeat and distribute afterward. However, we are using a library to try to identify malicious ones. Some recent supply chain attacks have shown us that we can never be too safe, and several packages in PyPi have been shown to be malicious. Even the extensive and well-used ML library "PyTorch" was recently in 2023 found to be compromised[4]. However, for such large packages, these attacks tend to be detected quickly due to the number of people using and relying on them. One of the precautions taken when using these libraries is to ensure the name is correct and that one of the newest but well-tested versions is used.

For deciding on what ML models to use in the experiments, inspiration has been found among previous research papers and by looking at the most commonly used models. Another factor was whether it was included in the SciKitLearn package, which suited our needs and would ease the process by relying only on one library for the ML part.

Based on previous research on using ML for detecting malware, some of the most common models used are from the model categories known as statistical, distance, and rule-based methods. The papers by Shalaginov *et al.* [20], Singh *et al.* [25] and Chio and Freeman [8] describe some of the most well-known and used ML approaches in depth. How to optimize and get the best performance for each model is out of the scope of this thesis as we are more interested in whether it can perform sufficiently when implemented with minimal effort. To answer RQ3 a few models from each category are implemented. Hence, it is possible to see what category and type of models are best suited for such a task as classifying static indicators. This will also partly contribute towards answering RQ2 in identifying the best-performing features. How will be described further in Section 3.2.3. The data that will be used are labeled, so the thesis focuses on the "supervised learning models" as previously described in Section 2.1.4.

The models which are used in the thesis will be mentioned below but more detailed descriptions are to find in Section 2.1.4 From the distance-based category will Support Vector Machine (SVM) and Multi-layer Perceptron (MLP) be included. Both are highly common to use and represent both simple and advanced types. The models are also mentioned as SVC and MLPC in the thesis because of the name they have in the Scikitlearn library being classifiers.

For the rule-based category of methods to use are some variations of tree-structured models. Decision trees, RandomForest, and Gradient Boosted trees will be included in the project. As described in Section 2.1.4 they are based on the same structure but with some variations for optimizing. This is interesting to look at to see whether it impacts the results.

The last category of methods to be included in the statistical model is Naive Bayes. It utilizes a simplistic approach by relying on statistics discovered in the data to predict. Since it had proven to perform well in real-life scenarios in some cases, it was interesting to include it in the comparison.

To verify the results we find some of the experiments will be run with cross-validation using the K-fold method. It will be used to both verify the results and for splitting the datasets into train and test parts. The second method that also will be applied in some of the experiments is the included function called "train-test-split". This method allows us to say how large of a test set should be created in percentage and it will automatically randomly divide the dataset into two parts of the desired size. The reason for using both methods is that cross-validation retrains the model K times with different parts of the dataset and therefore takes more time and resources to run. This is evaded in the second method as well as we get one model from the training process instead of the number for K.

3.1.2 Metrics

Scoring metrics are an essential part necessary to compare results. There are several to choose from and which ones to pick and why will be discussed in the following subsection of the thesis.

When conducting experiments with ML models, there is a need to be able to evaluate the results that are achieved. One standard method to display the results is the confusion matrix. It displays the number of false positives, true positives, false negatives, and true negatives in a matrix, making it easy to compare results based on numbers found in each category. Since each metric has its own strengths and weaknesses, published articles do not always agree upon what metrics are preferred. However, they are all based on the confusion matrix, which is common to publish. We chose to do so. This way other researchers can calculate their preferred matrix for their own specific comparison. When discussing the results other standard terms to use are the type-1 and type-2 errors. Type-1 errors refer to the number of false positives, and in this case, this means incorrectly labeling benign packages as malicious. On the other hand, type-2 errors refer to the false negative rate. and means the rate at which malicious packages are classified as benign. Besides the scoring metrics are these terms often used to describe the results and different use cases could prefer to reduce one error type over the other.

Several methods and metrics can be used to describe the results, where one of the simplest and most common is **accuracy**. This method is computed by dividing the number of correct predictions by the number of predictions and then multiplying by 100. The positive about this method is it is easy to use and implement and could give the reader an impression of how the model performs. However, the weakness of this metric is that it does not consider whether it is more accurate in predicting one class over the other. When using imbalanced datasets, this could potentially lead to high-accuracy results, but all samples are classified as the majority class, and the model is not able to detect any of the minority class.

Other metrics that are more often used to give the reader a more complete picture of the results are **precision** and **recall**. Precision is the term that describes type-1 errors which refers to false positives. On the other hand, recall, also called sensitivity, measures type-2 errors and refers to the false negative rate. These two metrics are often combined in a method called **F1-score** which combines precision and recall. The score presents a balance between them which is beneficial for imbalanced datasets where one class is over-represented. This gives us a better view of how each class performs and gives a more realistic view of the results.

Some of the metrics for presenting the results of a classification problem are described, for the experiments in this thesis accuracy and the F1-score will be included among the results. Since they both are widely used in ML projects, they are easy to implement and give the reader a good impression of how the models perform and are to be used when comparing the results.

3.2 Data

The following section will describe the data being used in the thesis, firstly where it comes from, then what it contains, and how it is being processed before it can be used in the desired experiments.

3.2.1 Origin

The origin of the data is the collaboration with ReversingLabs, which already possessed a large number of packages extracted from the PyPi registry. As they continuously conduct research into malicious packages in open-source registries, they were able to provide data and expert knowledge for this research project. All packages they analyze are run through their TitaniumCore platform, a machine-learning hybrid cloud platform that scans through different files to find malware and other threats. Their platform can handle many file types and is built to be scalable and handle a large amount of traffic[26]. Based on the work done by Reversinglabs regarding supply chain attacks and security research into open-source repositories, this thesis is enabled by having access to a large dataset of both benign and malicious packages.

The data provided for this project was received in a 503MB large text file which contained metadata and behavioral information in the form of indicators of 382,712 samples, where 7,639 were labeled as malicious and 375,073 were labeled as benign. As there are many more legitimate packages to be found in the PyPi repository, the dataset is also quite skewed regarding how many samples are found in each class. To increase the number of malicious samples they were able to provide, they included all versions of the malicious samples they had stored. In contrast, only the latest versions of the benign packages were provided to keep the numbers somewhat reasonable for this project. Later in the thesis will the impact of a skewed dataset be discussed. It is realistic to have a lot more benign than malicious packages as most are taken down rather quickly, and most of the activity is legitimate. However, as will be discussed later in the thesis, some experiments will be conducted with various data compositions to see if the models improve. One reason behind it is that models need data to be trained to classify correctly. By varying the degree of imbalance, the trained model might be able to take the minority class into higher consideration. Regarding the time frame in which samples have been collected ReversingLabs have been conducting research into these open-source repositories for about four years. They try to mirror what is available in the live repository for conducting analysis. That means all the samples in the dataset have been found in PyPi at least for some time during the last four years until January 2023. Due to confidentiality, the dataset will not be publicly available.

As the thesis focuses on static indicators, all data received is the extracted indicators and parts of the metadata that is commonly found in the Python packages. Each sample in the dataset contains a certain amount of information in fixed fields. Having the file structured in a certain way makes accessing and reading individual data elements easy. In the example, the listing in 3.1 is the structure of one individual sample displayed. As seen, each sample is identified by its hash-sum, which is unique for each sample and makes it possible to pinpoint each precisely. Generally, for all samples, the metadata fields containing the project name, version, and package size are included as well as other features, most noticeably "threat name". This indicates what capabilities the samples have as they are divided into categories such as "Trojan", "Infostealer", "Potentially Unwanted Software" and other types that already are well established in the industry. There is, however, no identification as to what family it might belong to, but it could indicate behavior specific to each type. In this thesis, the field will be used to know and determine whether samples are malicious or benign.

Code listing 3.1: Sample format preview

```
"dad9821834bf916ddfc9e6eb79b4218c444c6057": {
    "project_name": "pip_security",
    "version": "0.0.8",
    "package_size": 5711594,
    "threat_name": "Trojan.Generic",
    "reviewed": "yes",
    "package_stats": {
      "python_files_count": 1,
      "executable_files_count": 1,
      "non_text_files_count": 1
    },
    "indicators": [
      "Creates a process.",
      "Executes a file.",
      "Exits the script.",
      "Imports the \"os\" module, which contains miscellaneous operating system
```

```
        interfaces.",
    "Imports the \"setuptools\" module, which is a package development process
        library.",
    "Imports the \"sys\" module, which provides access to system-specific
        parameters and functions.",
    "Writes data to the STDOUT stream."
  ]
}
```

Another feature that could be of interest is the package stats which describes the number of different files provided in the package, in the example, only one Python file is present. Besides the basic metadata is the list of indicators of high interest, these fields describe the static indicators found in the package and can give us an idea of what the package can do. The indicators, as seen in the example, describe different characteristics for each package, such as the module "OS" is imported or that it executes a file. These indicators are produced in the static analysis process and end up as a list describing what packages are included, the activity it executes, and the capabilities it might have. In the raw data, all samples contain a set of these indicators and during the preprocessing, all indicators are extracted and stored in a list numbered by the order they are found. The list of all unique indicators ended up at 571 numbered descriptions which are referred to as an indicator with a number. The content of each sample after converting each of the indicators seen in Code listing 3.1 can be seen in Table 3.1. After each indicator has been converted to a binary list, does it contain the number "1" if the mapped indicator is found in the sample. Some of the initial motivation for conducting this project is also based on whether it is possible to identify malicious packages only based on the type of data seen below and to understand better what combination of indicators is most likely to appear among the malicious samples.

Table 3.1: Visualized sample content and structure

hash	ac2625cd4d072143fc286d7ed8d6b634bf0470e7
project_name	sphinx-issues
version	3.0.1
package_size	8227
class_malicious	0
py_file_count	1
exec_file_count	0
non_text_file_count	0
num_indicators	3
indicator0	0
indicator1	0
..	
indicator570	0

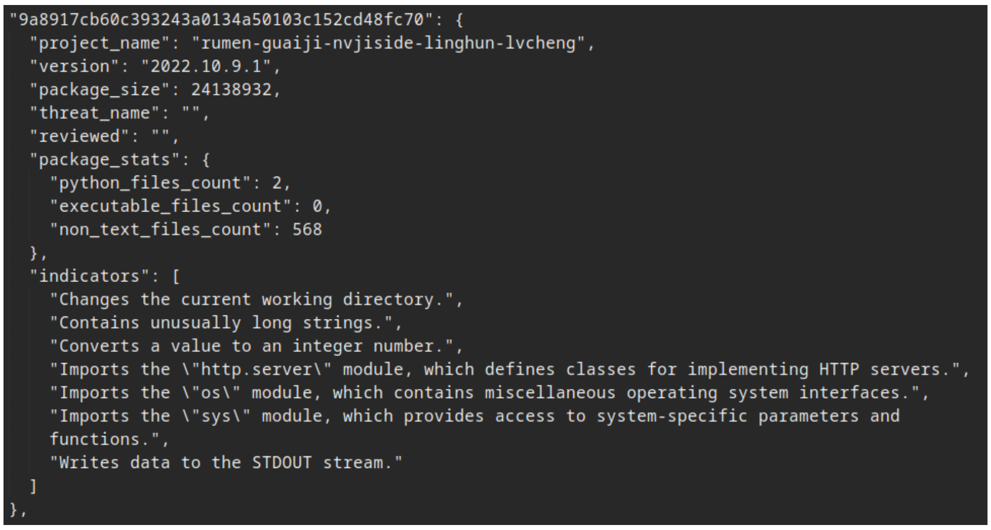
3.2.2 Pre-processing

Before the initial data can be used in ML applications and the planned experiments it is necessary to convert it to another suitable format. The most common file type to use is CSV which stands for "Comma Separated Values" which are easy to read and write to and from programs and do not need compression or indexing source. A short program is made in Python using Jupiter-notebook to load each sample and convert it to the CSV format, in the process also converting the indicator list to a binary list representing all the indicators found in the entire dataset. This is one method to make each sample a fixed length and make them all fit a specific format for more manageable handling using ML models and is chosen over others because of its ease of implementation as well as considering the number of unique indicators is manageable. Another method that could be of interest to try at a later stage would be Word2Vec combined with a common bag of words (CBOW) which is a method that can group words together based on their context. By using a different approach like this, similar indicators could possibly be grouped together, the number of features could be reduced as well as each feature might be of higher quality. It could also possibly be more future-proof in regard to handling new samples with new indicators not previously seen as the chosen approach would not be able to handle new and different features whiteout recomputing the whole model and pre-processing method or removing the new ones.

A couple of other modifications are also done on the data during the pre-processing phase, one being to remove features and another converting to numbers. From the example seen in listing Code listing 3.1 is the field "reviewed" removed as it does not provide any useful information to the model other than it has been reviewed by a person. In a certain scenario, it could be used for identifying how many samples have been manually verified and not, and then have more confidence that the classification is correct. As the time is limited we assume that the data is mostly accurate and the other features could be depended on. Another feature that is removed is "package-stats" which contains three features packed inside a list which is then extracted and used as individual parts in the sample.

Another technique that will be applied to the datasets before they can be trained or tested is normalization. A part of the preprocessing steps in ML is to normalize the data, especially when having features with numbers in different scales. The function StandardScalar from Scikitlearn[27] allows us to normalize the numeric features by removing the mean and scaling for each feature based on the whole training dataset. By using this process, the models are better suited for comparing features as they are to be found in the same range of numbers and can make better predictions. When using the function to find the mean and standard deviations for each selected feature in training data, the values stored so the same can be used for the training set, which makes the results more realistic compared to doing the normalization processes for each individual dataset.

Another major decision is taken on how to handle version numbers in the dataset. As identified by related work, dependency confusion is a common attack



```
"9a8917cb60c393243a0134a50103c152cd48fc70": {
  "project_name": "rumen-guaiji-nvjiside-linghun-lvcheng",
  "version": "2022.10.9.1",
  "package_size": 24138932,
  "threat_name": "",
  "reviewed": "",
  "package_stats": {
    "python_files_count": 2,
    "executable_files_count": 0,
    "non_text_files_count": 568
  },
  "indicators": [
    "Changes the current working directory.",
    "Contains unusually long strings.",
    "Converts a value to an integer number.",
    "Imports the \"http.server\" module, which defines classes for implementing HTTP servers.",
    "Imports the \"os\" module, which contains miscellaneous operating system interfaces.",
    "Imports the \"sys\" module, which provides access to system-specific parameters and
    functions.",
    "Writes data to the STDOUT stream."
  ]
},
```

Figure 3.1: Showcasing how version numbers can appear.

vector and means setting the version number artificially high for the system to prefer the malicious package over legitimate ones which then contain a lower version. That is one of several problems that can be observed in the data from manual inspection, other issues can be that text, such as the project name is written in the version number as well and is so a text string instead of a number. As some of the models that are going to be used only handle numbers make this a problem. One of the simpler solutions that are being used in this thesis is to use code for automatically detecting numbers in the version field and adding them together to have a single number to compare against. It is probably not an ideal solution but since the project focus is more on indicators and the short time aspect makes it a viable solution for this thesis. It should however work well for most packages where the version numbers are of normal sizes, for packages where text and version numbers are combined, and where the version is fixed at a high level. Some errors must be expected and as seen in the Figure 3.1 legitimate packages can also contain abnormal version numbers and reduce the integrity of the feature itself, which will be taken into consideration later in the thesis when looking at the results. As some of the models are only able to handle numbers the hash and project name are also removed later in the process right before the models will be run, but it is kept in the datasets to easier conduct further analysis into specific samples if needed.

3.2.3 Datasets

In preparation for conducting the experiments, several different datasets will be created based on the main data. By creating multiple datasets with different compositions of data, it is possible to see how specific algorithms are able to handle and perform with different amounts of samples, especially for this dataset that has a big difference in the number of samples from the two classes of benign and malicious.

Before dividing data into different datasets for use in experiments 20% of the initial data are randomly selected and saved. The reason for removing a part of the initial dataset is to have some untouched data that can be used as a verification dataset to test the models at the end of the project when the best settings are found and applied. The results can then be verified by testing against "new" and unseen data. This will onward be called the verification dataset and is removed before the rest of the datasets are created.

The datasets that will be used in the experiments can be divided into three groups based on methods to reduce the imbalance in the dataset. A couple of techniques that can be used are down-sampling and up-sampling, while the last category of dataset types will consist of more realistic approaches to what is most likely seen in the wild and the received dataset. By comparing the results from the different datasets, it is also possible to see how different models perform based on how much data they are fed. More explanations about the different categories of datasets will be described in this section.

For conducting the experiments and to answer the RQs are a few different techniques applied when creating the different datasets. The idea behind this was to see how the models performed with different amounts of samples as well as the degree of imbalance. It would also possibly allow us to see whether some selection of samples gave a noticeably better or worse performance. Each dataset can be grouped into three different categories based on what technique it is based on. The first technique for handling imbalanced datasets is to use down-sampling. By reducing the number of occurrences from the largest class the dataset can become better balanced and make it easier for the ML model to correctly classify the minority class. Another well-known technique for reducing the imbalance in datasets is to use up-sampling. With this method samples from the minor class are duplicated to reduce the imbalance in the dataset, without reducing the information located in the major class, it improves the balance and betters the significance of the minor class. The downside of duplicating the minor class is the reuse of the same samples, which could reduce the generality of the model depending on how well the samples reflect the real world. Besides trying to balance the dataset as seen in the other dataset versions, a few do appear as possibly more realistic than others. Some of the datasets utilize the complete range of benign samples but also increase the number of samples from the malicious pool as well. Even though the full dataset does not necessarily mirror reality in what can be expected for the different classes, a few of the datasets will be set up to be similar. It means quite an imbalanced dataset which is interesting to include for the sake of comparing results and checking whether the models can correctly classify the minor class even though they see so few samples, which is true for real-world applications. Each dataset will be given a brief introduction below but an overview of the differences between them in regards to the number of samples can be seen in table Table 3.2. When referring to the full dataset below describing each of the datasets used in

Table 3.2: Overview of the datasets used in experiments

Dataset	Number of samples	Benign	Malicious
Dataset1 (Dataset1)	20,414	14,291	6,125
Dataset2 (Dataset2)	12,250	6,125	6,125
Dataset3 (Dataset3)	9,187	6,125	3,062
Dataset4 (Dataset4)	306,170	300,045	6,125
Dataset5 (Dataset5)	162,272	150,022	12,250
Dataset6 (Dataset6)	318,420	300,045	18,375
Dataset7 (Dataset7)	21,437	18,375	3,062
Verification dataset	76,542	75,028	1,514

the experiments is the reminding data after the test dataset is extracted for having untouched data to compare against in the end.

Dataset1: This is in the category of down-sampling and utilizes all of the malicious samples which equals 30% off all the samples in the dataset. The other 70% percent of the dataset is benign samples which were extracted from the full dataset.

Dataset2: This dataset consists of all the malicious samples while the number of benign is reduced to the same number as for benign, using the down-sampling method. For this dataset, the split between malicious and benign is equal and will be used in comparison with the other datasets and with different models to identify whether the results change with a balanced dataset.

Dataset3: For the third dataset which is also utilizing the down-sampling technique 50% of the malicious samples are extracted from the main dataset together with the doubled number of benign samples. This equals 3,062 malicious and 6,125 benign in the dataset, making a total of 9,187. By making a separate dataset taking a selection of the minor class is to see whether the results change when randomly selecting samples since it contains some bias because of different malware campaigns or similar that could populate a portion of the dataset.

Dataset4: This is one of the more realistic approaches and uses all the data except the validation data left out in the beginning of the project. That results in quite an imbalanced dataset which is interesting to include for the sake of comparing results and checking whether the models can correctly classify the minor class even though they see so few samples, which is valid for real-world applications as well.

Dataset5: One of the datasets which are created with the up-sampling method, is created by duplicating all the existing malicious packages which means two times the number of malicious samples, which equals 12 250. The amount of benign, on the other hand is down-sampled and uses 50% of all the samples in that category and equals to 150,022.

Dataset6: The second dataset uses up-sampling, instead of doubling the number of features it has now been tripled which equals 18,375 malicious samples. For the number of benign samples, all are used, for it to be more realistic regarding the total dataset that has been used.

Dataset7: Like dataset3, this dataset also uses 50% of the malicious packages but differs from the other variation in the number of benign samples selected. For this version, the number of benign samples is six times higher than the 3,062 malicious samples extracted and equals to 18,375.

Verification dataset: As previously mentioned, this dataset is created by extracting 20% of the initial data before this process of dividing the data into different datasets is conducted. This set of samples is then saved until the end of the project before it is used to verify the trained models.

3.3 Experiments

After describing what technology and data are being used, it is time to go through how they are combined into the experiments, which are split into five parts. These experiments can again be seen as two parted where experiments 1 and 2 are seen as the first part of experiments since these results will be utilized in experiments 3 to 5. In experiment 1, the features will be ranked to identify who gives the most information in separating malicious and benign samples. This experiment will be used to help answer RQ 1 and 2. Experiment 2 is used to implement and test the selection of the most common ML models described in Section 3.1.1 and compare the results against each other. The second group of experiments consists of the last three, named experiment 3, 4, and 5. These will utilize some of the results achieved in the first group to conduct some more detailed analysis experiments. Experiment 3 is called model optimization and will test three of the top-performing models from experiment 2 and test a few parameters for optimizing the model. In this experiment will also be the threshold for reducing one of the error types be looked into. This experiment is used to help answer RQ3 by looking at what performance can be expected from the models. Experiment 4 will be used to find the best combination of features to use in further classification. This will be done based on the results found in experiment 1 to identify how few features can be used without a significant drop in performance. Experiment 5 is the last one and will be used to test the results achieved in the first four experiments against a verification dataset. Besides re-testing the models will also some statistics for the dataset be looked at to answer RQ1

Before describing how the first experiment is to be conducted is there a parameter that needs to be mentioned. One option which will be used during most experiments is the random state. While conducting experiments 1 and 2 and during the prepossessing of data, will all randomization options use the parameter "random-state" of 42. This means when datasets are split into training and test sets or when training the model, a number is used to calculate the operation’s randomness. All operations that are supposed to be random use a logic that calculates the randomness based on something specific in this case a number, which makes these operations "pseudo-random". Besides the possibility of reproducing the experiments and results achieved in this project, it will also make all operations more equal when testing the different models as changing the random state might give different results across the models. The "random-state" is also changed to seven in experiments 3, 4, and 5 to see if this affects the results. If time and resources were available would it also be interesting to redo multiple of the tests using different numbers as the "random-state" to see whether it impacts the results, this is considered future work

3.3.1 Experiment 1: Feature ranking

The first experiment conducted is to look closer at how the features perform. The feature ranking method "information gain" or IG for short will be utilized to understand the most helpful features and contribute to answering RQ2. IG is used to calculate the reduction in entropy by dividing the dataset by specific features. The method is used to describe how likely a feature is to appear and the entropy can be interpreted as how much information is provided for each feature. The higher entropy, the better the feature at dividing the dataset into different groups. This allows us to rank the feature and get an overview of what features are better or worse to use in the experiments as the more features, the more complex and more resources are needed to compute. IG is computationally inexpensive compared to other methods making it a fast and easy way to rank features. This makes it suitable for these initial experiments. Still, with a more extensive scope and the time to go more in-depth, several of the other methods available should also be implemented to compare the features against each other.

The experiment is conducted on all seven datasets shown in Table 3.2 where the algorithm IG is tested on each individually, and the top 10 highest ranked features will be extracted from each dataset and compared against each other to show whether the same results can be seen when the number of samples varies as they do in the different datasets. By ranking the features, we prepare for what is to come in the following experiments as well as identify what is seen as the most valuable features which relate to RQ2.

One note about the last part of the results is that all of the datasets are normalized together. Different from how the process is usually conducted where the training data is normalized first, and the test data is normalized based on results from the training data. Here the whole dataset is normalized together before the feature ranking algorithm is run.

3.3.2 Experiment 2: Model comparison

For the second experiment in the first group, will all seven datasets be tested with the selection of ML models described in Section 3.1.1 in order to see how they perform on different types of datasets with different degrees of imbalance. The reason is to help answer RQ3 by improving knowledge about what models work better than others on the data type used in this thesis and for different amounts of data. An in-depth comparison of different models is out of scope so a selection is chosen and tested with default settings to indicate what performs best with little to no optimization. Different algorithms do perform differently based on the situation and are designed for different use cases. Getting a brief overview of how some of the most well-known models perform is beneficial in the search for finding the best features, models, and information to identify malicious packages. This experiment will be conducted twice using cross-validation with two different numbers for K, 5, and 10. The reason for picking these two numbers is based on what is commonly seen in tutorials as well as them being reasonable in terms of how large the testing set will be. When using the number 5 the size of the test set when splitting each dataset equals to 20% but when 10 is used the size of the test set decreases to 10% of the data.

3.3.3 Experiment 3: Model optimization

In the second group of the experimentation process, in experiment 3 will some deeper analysis be conducted. This experiment is based on the results from the first two experiments conducted. The three best-performing models identified in experiment 2 are retested on the datasets trained with a few different parameters, both to verify the results and to determine if they could be improved even more by applying small changes to the configuration. The purpose of this experiment is to answer the RQ3 regarding how well these models perform and give an indication of which model has more potential than others. As each model is of a different type, do they not have the same parameters which can be used and adjusted. The different parameters used in this project will be briefly described before the results are presented in the next chapter. Each action in this experiment uses dataset 1 as it contains fewer samples than the others. It is not as realistic as some of the others in terms of the number of samples or the composition, but it will still give an indication of what performs better. It also drastically reduces the time and resources needed for testing each option. The thesis focus is not training the best models but rather on giving an overview of what could be expected. A more thorough test across multiple datasets with more parameters would be highly interesting but is considered future work.

Thresholds

Another step in experiment 3 about the model optimization process is to look at how the models can be altered to improve either the precision or recall results. Depending on the use case improving the model results in one of the metrics can be preferred, in a setting where the lowest FP rate is wanted being able to tune the model can be of interest. As stated in Vu *et al.* [2], the maintainers prefer a system with close to or zero false positives as they are only able to handle a few incidents and they prefer letting multiple malicious samples pass but rather catch most low-effort attacks. This can partly be confirmed by experts in the industry as well as they often don't have the manpower to analyze large amounts of false positives, and it will be more economical to detect a possible attack with other means. Two methods for investigating what thresholds can be set for the models to prioritize is a Receiver Operating Characteristics (ROC) curve and Precision-Recall (PR) curve. In order to implement these two measuring techniques the ML models need to be able to provide probability scores when applied to the test data and not only return binary results to what class is correct. The point of evaluating the models with these techniques is to see what thresholds can be used to favor one metric or find the optimal balance between precision and recall, or TPR and FPR. The ROC curve creates a graph plotting the model performance at different thresholds for the TPR and FPR. Since the thesis mainly deals with imbalanced datasets, will the use of a similar method PR curve be more beneficial as it is able to give more information[28].

The PR curve will be created for each of the datasets with the three selected models. By doing the experiment on all the datasets is it possible to compare the results in order to see how different thresholds would impact the results based on the number of samples and which methods the datasets are based on.

3.3.4 Experiment 4: Optimal number of features

In the search of identifying the best-performing features and to see what impact a reduced number of features have on the models a new round of tests will be conducted. For each dataset, the results from experiment 1 will be used to test the three top models identified in experiment 3. Based on the ranked features the worst will be removed one at a time before each of the models are tested to see how the performance is impacted. It will also allow us to see how few features can be used and still maintain a certain performance. By running these tests, we will also attempt to verify the results from the feature ranking by seeing how well the top features perform by themself or whether more features are better for the results.

The experiment is conducted by going through each of the features in the ranked list from the worst to the best. For each feature are the three models trained and tested using cross-validation 5-fold where the average F1-score of these folds are used to compare the results against each other. This experiment will only be conducted on dataset 1 to see what results can be achieved how the results change based on the number of features. As each sample contains more than 500 features would the time and resources needed to test the model over 500 times be greater than what potential results it might reveal. For this thesis, the subset of samples is seen as sufficient, and deeper testing and experimentation with the number of features is seen as future work.

Another way this experiment could be conducted is to remove exactly the same features for the datasets to make the tests more similar. This might have an impact. However from a brief comparison, the top features seem to be quite similar, some features might change some places up or down the list based on what dataset. It introduces more "variety" in the test by doing it this way. Still, it will be used since it seems more realistic regarding the natural order to conduct ML projects, where feature selection is a natural part of the process. Computing different feature rankings and seeing that many of the same features are ranked as the top ones makes it clearer that they are helpful and provide some value across the different compositions.

3.3.5 Experiment 5: Model evaluation

After the project has gone through the first four experiments, the achieved results will be utilized in this last experiment. The results regarding what models and options to prefer should be identified and implemented before the final experiment is conducted. To finalize the thesis and test whether the discovered results might have some value, will the three best models be tested on the verification dataset that was left out before creating the datasets for the experiments. The sole purpose of leaving out that portion of the dataset, in the beginning, was to have the ability to evaluate the models created on data not previously seen in the thesis. This allows us to see how well the trained models can handle new data.

This experiment will be used as the final means of answering RQ1 and RQ3, it will however also provide some verification of the results regarding RQ2. Firstly will this experiment build upon the results achieved in Experiment 4 by using the optimized models against the verification dataset. Each of the seven datasets will be used in training the three selected models found in experiment 2 with the settings discovered in experiment 3. This testing process is then conducted twice to test the models and datasets against the verification data using all indicators and one time only using the top 100 as identified in experiment 1. After running these experiments, should we be able to further answer RQ2 in terms of how well the top 100 features work in classification. It will also contribute towards answering RQ3 by giving an insight into what performance can be expected from the models used in this thesis.

After conducting these experiments on the verification dataset, will the miss-classified samples be stored for each test and used to compare statistics. We will then look at what features are commonly found in the entire dataset, among the correctly classified and the wrongly classified samples. By looking into the differences between the whole dataset and the miss-classified samples some indicators might point out as the reason why it evaded the model and give an indication of what could be done to improve them. These statistics will also help in answering RQ1. Other results that are interesting to look into are whether the same samples are missed in the different datasets or if the models tend to overlook the same samples. This will simply be measured by comparing each class's missed samples with the number of unique samples. Another experiment that would be interesting to look closer at is to dive deeper into the results and samples missed and see what type of characteristics it has and look at the package types to improve the knowledge even further, but this is seen as future work.

Chapter 4

Results

After describing the methodology in the previous chapter, the following presents the results discovered in the experiments. Similar to the method's structure, the results will be divided into five experiments. Experiments 1 and 2 can be seen as the first group, and these results are used to decide on what to use in the last three. The first two show more general results from ranking the features and testing the selection of models. In contrast, the second part which consists of experiments 3, 4, and 5 goes more in-depth into the three top models identified in the first. After the three models are selected, a few optimization steps will be tested and implemented to see if the results improve from the first part before testing again against a new test dataset. Before describing the results, Table 4.1 contains an overview of the indicators listed in the thesis besides the non-binary features. Each feature is numbered and accompanied by a description, and the table will be used when referencing and discussing what each indicator means. Each indicator is found by using static analysis techniques to identify the capabilities and characteristics of a sample, how it is found and created is described more in Section 3.2.1.

As previously described the experiments are divided into five parts, and the results from each will be presented in their own section below.

4.1 Results experiment 1: Feature ranking

The results from the feature ranking are presented in Table 4.2 and show the top 10 ranked features from each of the seven datasets described in Section 3.2.

When looking at the results in Table 4.2, we can see that several of the same indicators tend to appear in most of the datasets. Similarities can be seen based on how the datasets are constructed, such as for datasets 1, 2, and 3, which used downsampling. They only differ in the composition of benign and malicious samples and the number of features. They have almost identical results except for a few deviations, the order of a few and Indicator39 found in dataset 2, while the two other datasets listed Indicator46 instead. When looking at dataset 4, which contains all the features, and datasets 5 and 6, which have increased the number of malicious by duplication, the same indicators are present but have some

Table 4.1: Overview of indicators and descriptions.

Indicator	Description
Indicator4	Concatenates an unusual amount of strings, commonly used for obfuscation.
Indicator9	Creates a process.
Indicator12	Creates/Opens a file.
Indicator22	Imports the "os" module, which contains miscellaneous operating system interfaces.
Indicator27	Imports the "setuptools" module, which is a package development process library.
Indicator39	Reads from files.
Indicator45	Uses string related functions.
Indicator46	Writes data to the STDOUT stream.
Indicator77	Imports the "requests" module, which is used for sending HTTP requests.
Indicator80	Makes HTTP GET requests.

Table 4.2: Top 10 highest-ranked features for each dataset

	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Dataset 6	Dataset 7
1	package_size	package_size	package_size	package_size	package_size	package_size	package_size
2	py_file_count	py_file_count	py_file_count	indicator12	py_file_count	py_file_count	py_file_count
3	num_indicators	num_indicators	num_indicators	indicator27	num_indicators	num_indicators	num_indicators
4	indicator4	indicator80	indicator80	num_indicators	indicator4	indicator4	indicator4
5	indicator80	indicator4	indicator4	py_file_count	indicator80	indicator12	indicator80
6	indicator77	indicator77	indicator77	indicator46	indicator77	indicator80	indicator77
7	non_text_files	non_text_files	non_text_files	indicator22	indicator12	indicator77	non_text_files
8	indicator12	indicator12	indicator12	indicator4	non_text_files	indicator27	indicator12
9	version	version	version	indicator45	indicator46	indicator46	version
10	indicator46	indicator39	indicator46	indicator39	indicator45	indicator45	indicator46

differences from the first group. There do not seem to be significant variations in terms of what features are listed, but the order they appear tends to change. A reason might be that the number of malicious samples changes the dynamic in the dataset. It could also be because of the composition of samples, especially in the down-sampled variations where much information is lost when the number of benign samples is reduced.

One aspect that makes it hard to detect malicious samples based on single indicators is that they could likely appear in legitimate applications, and most indicators are not necessarily malicious. Indicators 46, 77, and 80, which have a description in Table 4.1 indicate the ability of a sample to write output from the program and make GET web requests over HTTP. That could mean the specific sample requests additional information online or tries to download additional malware. Even though other indicators, such as Indicator4, which reacts on unusual amounts of a string concatenation, could be viewed as more malicious since it could be used as an obfuscation technique, but just as well for legitimate purposes. When looking further into what features to use to detect malicious samples, should it be looked closer at what combinations of features are commonly found in the malicious packages. A deeper analysis of the combinations found in the malicious samples is seen as future work but could provide more knowledge on what specific feature combinations could indicate malicious packages.

4.2 Results experiment 2: Model comparison

The following section will present the results from testing a selection of models described in Section 3.1.1 against the seven datasets shown in Table 3.2. Two tables are shown, the first Table 4.3 showing the average F1 score from the cross-validation 5-fold technique. The second table that can be viewed is similar to the first but shows the results when using 10-fold for cross-validation. The last entry for both tables consists of the average F1 score across all datasets. A brief description of the models can be found in the Section 2.1.4, but to clarify the shortened names: SVC stands for Support Vector Machine, KNN for K-Nearest Neighbors, GB for GradientBoosting and MLPC for Multi-Layer Perceptron Classifier.

Table 4.3: Results model comparison average F1-score using 5-fold cross-validation

Model	RandomForest	DecisionTree	SVC	Naive Bayes	KNN	GB	MLPC
Dataset 1	0,76	0,46	0,96	0,59	0,94	0,53	0,98
Dataset 2	0,77	0,66	0,97	0,77	0,95	0,59	0,98
Dataset 3	0,78	0,50	0,96	0,60	0,95	0,43	0,97
Dataset 4	0,71	0,11	0,88	0,06	0,90	0,32	0,91
Dataset 5	0,74	0,41	0,93	0,21	0,92	0,43	0,94
Dataset 6	0,64	0,35	0,93	0,16	0,92	0,38	0,94
Dataset 7	0,82	0,44	0,94	0,36	0,93	0,46	0,95
AVERAGE:	0,75	0,42	0,94	0,39	0,93	0,45	0,95

Table 4.4: Results model comparison average F1-score using 10-fold cross-validation

Model	RandomForest	DecisionTree	SVC	Naive Bayes	KNN	GB	MLPC
Dataset 1	0,93	0,55	0,96	0,59	0,95	0,58	0,97
Dataset 2	0,90	0,61	0,97	0,77	0,96	0,64	0,97
Dataset 3	0,81	0,48	0,96	0,60	0,96	0,51	0,96
Dataset 4	0,64	0,12	0,88	0,06	0,90	0,33	0,91
Dataset 5	0,73	0,41	0,93	0,21	0,92	0,42	0,94
Dataset 6	0,74	0,39	0,93	0,16	0,92	0,45	0,92
Dataset 7	0,83	0,43	0,94	0,36	0,93	0,52	0,95
AVERAGE:	0,80	0,43	0,94	0,39	0,93	0,49	0,95

The results from the second experiment about model comparison can be seen in the two models Table 4.3 and Table 4.4. The first observation that can be seen is that the average results for the seven datasets are the same for each model in both tables except for RandomForest, DecisionTree, and GB. RandomForest has the highest difference in performance between the two tables, with a score 0.5 higher when 10-fold is used. For DecisionTree, the average score only changes by 0.1, while it changes by 0.4 for GB. Except for these changes, the average of the results stays the same for both tables. This can show that the number of splits used during k-fold cross-validation has less to say, at least reading from these results. The reason might be how the dataset is structured and what samples are included since the difference using a larger number for K-folds means splitting the dataset into smaller pieces and reducing the dataset size to be tested. Too small of a test set could lead to an unfortunate combination of samples, which lowers the overall score. Still, since the test is run multiple times with each split piece as the testing set, it will be possible to determine whether one run has significantly lower scores.

For the best-performing models, we can see SVC with an average of 94%, KNN with 93%, and MLPC with 95%. These scored significantly higher than others such as Naive Bayes, GB, and DecisionTrees, with average scores of 45% and lower. Only RandomForest had an average score that could be considered good at 80% but is still quite a bit lower than the three at the top. The lousy performance could be because the models are unsuitable for this type of data, or it would have needed a more comprehensive preprocessing and feature extraction process to perform. Other reasons could be the need for further optimization and different parameters for the models to perform in this scenario. These results are also used to decide on what models to use in the following experiments. It also contributes towards answering RQ3 by showing what performance can be expected from the selected models.

4.3 Results experiment 3: Model optimization

As stated in the methodology, the three top-performing models from experiment 2 will be used in this and the following experiments. This experiment was conducted to test a few parameters to identify whether modifying some default options would pay off in terms of increased performance. As found in the previous subsection could we see the two tables Table 4.3 and Table 4.4 that the average results for each model across all datasets are pretty similar with different values using K-fold. There are also three models that stick out with significantly higher average scores when compared to the rest, they are MLPC, SVC, and KNN. When looking at the average F1-score across all datasets, does SVC score 94%, KNN 93%, and MLPC 95%. These three models will then be used in the following experiments, which in this will be about testing a few different parameters before looking at the threshold using the PR curve. Each of the three models has its own subsection where the parameters that will be used are briefly described.

4.3.1 MLPC

One of the models that performed best in experiment 2 was MLPC which gave an average score of 95% seen in Table 4.3 across the seven datasets used in the initial testing. MLPC, a classifier based on a neural network, is implemented with many options to choose from. When optimizing the model, the user can choose the number of hidden layers to use, what loss function to implement, and many others for more granular control. This experiment will test the options for deciding what activation and solver function to use. The activation function is used to make the network more dynamic and able to handle more complex tasks and learn by implementing mathematical functions that can map the complex output from the different nodes in the network[29]. Each method has different properties that make them more suitable for specific types of problems, the ones available in ScikitLearn are called "identity", "logistic", "tanh" and "relu". The second parameter tested is the solver function which refers to the weight optimization between the nodes in the neural network. The options available are named "lbfgs" which by SciKitLearn is said to be a "quasi-Newton method", the second option is "sgd" short for "stochastic gradient descent". While the third is "adam" a gradient-based optimizer proposed by Kingma and Ba [30][31]. It was decided not to look closer at optimizing the number of hidden layers. This is one of the key parameters for this type of model, but it was considered too big of a task when the focus is to get an overview of what performance could be expected. As there is no standard way to find the optimal number, the range of options to test is seen as too large and therefore is added to future work. The default value used in Scikitlearn is 100 for the number of hidden layers.

The results from testing the options mentioned above can be seen in Table 4.5 which show what parameters are used, how each run of the 5-fold cross-validation performs, and the average of those five. A second table shows some extended results from how this model performs with the default parameters across the seven models. It also contains the number of both correctly and wrongly classified samples in the categories of benign and malicious.

Seen in the Table 4.5 is the overview of results found when testing the different options as described above. Two combinations stand out from these results with an average F1-score of 98%. The default options are "relu" and "adam", together with "logistic" and "adam" performed the best among the options. From these results on this type of data, whether the default parameters are used or not when training the MLPC model does not seem to matter. For the other options, some combinations scored significantly lower with an average score of around 90%, showing it does matter which parameters are chosen. As these experiments are conducted on dataset 1 the results are higher than what can be expected for higher dataset sizes since dataset 1 only contains a subset of the benign. Another aspect is the number of hidden layers which as mentioned previously in this section should have been experimented on but is put in future work. However from these results and what is found in experiment 2: model comparison the default value of 100

Table 4.5: Model optimization results for MLPC, random_state = 7

Model: MLPC						
Parameters	K1	k2	K3	K4	K5	Average
activation='identity', solver='lbfgs'	0,94	0,95	0,95	0,94	0,94	0.94
activation='identity', solver='sgd	0,92	0,93	0,93	0,93	0,92	0,93
activation='identity', solver='adam'	0,95	0,95	0,95	0,95	0,93	0,94
activation='logistic', solver='lbfgs'	0,97	0,93	0,97	0,96	0,94	0,96
activation='logistic', solver='sgd'	0,89	0,90	0,89	0,91	0,89	0,90
activation='logistic', solver='adam'	0,98	0,97	0,97	0,98	0,97	0,98
activation='tanh', solver='lbfgs'	0,95	0,93	0,95	0,94	0,94	0,94
activation='tanh', solver='sgd'	0,93	0,94	0,93	0,94	0,93	0,93
activation='tanh', solver='adam'	0,98	0,93	0,97	0,97	0,96	0,96
activation='relu', solver='lbfgs'	0,91	0,94	0,94	0,93	0,92	0,93
activation='relu', solver='sgd'	0,95	0,95	0,95	0,95	0,95	0,95
activation='relu', solver='adam'	0,98	0,96	0,98	0,98	0,98	0,98

Table 4.6: Results and time it takes for MLPC to be trained and tested with the train-test-split function.

MLPC	DS1	DS2	DS3	DS4	DS5	DS6	DS7
Accuracy	0.987	0.982	0.979	0.996	0.992	0.993	0.988
F1-score	0.978	0.983	0.968	0.916	0:653	0.946	0.958
TN	4246	1748	1832	89921	44751	89544	5495
FP	39	36	31	68	269	475	45
FN	40	27	26	231	83	128	30
TP	1800	1864	868	1631	3579	5379	862
Time	0:01:48	0:00:54	0:00:38	0:10:03	0:09:07	0:14:42	0:01:09

hidden layers performs quite well being one of the best models in this comparison.

Another table Table 4.6 does also show some extended results from how the model performs with its default parameters. In these results are the function "train-test-split" used to extract 30% of the dataset and show the confusion matrix as well as the F1 score for the seven datasets. It does also show the time it takes to train and test the model. This shows that on larger datasets like dataset 4 (DS4) that it takes 10 minutes to train and test the model.

4.3.2 SVM

The second of the three top-performing models is SVC which had an average F1-score of 94% across all datasets. The SVM model implemented in Scikitlearn allows for a large degree of customizability in regard to what options to provide and what thresholds to set for the model. One primary attribute is the kernel which is picked for this experiment to give an indication of what might work best for this type of data and give a foundation for further research into optimizing the more specific parameters to see what works best. The kernels available for use in this model are: "linear", "poly", "rbf" and "sigmoid. When talking about SVM, the kernel is the name for what algorithm is used to solve the problem, the features are plotted, and in this use case, a line is drawn between the features for being able to make a decision as to what to predict. The simplest method to use is when a line could separate the data and refers to the "linear" kernel. When the data is more complex and is not separable by only drawing a straight line, more advanced mathematical expressions can be used in an attempt to best separate the data.

The SVM model implementation used in these experiments is called "SVC" and is one of several variations that can be implemented from the ScikitLearn library. As specified in the documentation another implementation called "LinearSVC" is also recommended besides the one used here. Its different implementation makes it able to better handle larger amounts of samples as it offers more flexibility in regards to choosing a loss function and penalty[32].

Table 4.7: Model optimization results for SVC, random_state = 7

Model: SVC						
Parameters	K1	K2	K3	K4	K5	Average
kernel='linear	0,95	0,95	0,95	0,95	0,94	0.95
kernel='poly'	0,89	0,91	0,89	0,91	0,90	0,90
kernel='sigmoid'	0,69	0,83	0,32	0,70	0,32	0,57
kernel='rbf'	0,96	0,96	0,96	0,97	0,96	0,96

Table 4.8: Time it takes for SVC to be trained and tested with the train-test-split function.

SVC	DS1	DS2	DS3	DS4	DS5	DS6	DS7
Accuracy	0.977	0.974	0.971	0.995	0.989	0.992	0.980
F1-score	0.962	0.975	0.955	0.873	0.926	0.931	0.927
TN	4248	1723	1848	89985	44879	89916	5503
FP	37	61	15	4	141	103	37
FN	99	34	63	414	380	614	88
TP	1741	1857	831	1448	3282	4893	804
Time	0:00:23	0:00:10	0:00:04	1:14:03	0:09:08	1:04:12	0:00:16

The results from testing the different kernels in the SVC models can be seen in

Table 4.7. Similar to the results found for MLPC it is also here the default settings that achieve the best results. With an average F1-score of 96% does "rbf" score the highest above the "linear" kernel which also scored relatively high with an average of 95%. Conveniently does there not seem to be necessary to change the major parameter to improve the results, however, all models do have multiple options that can be modified and with the goal of creating the best models deeper testing is necessary. Dataset 1 is also used in this experiment and from what can be seen in the results from experiment 2 does that dataset have overall higher scores compared to the other where more benign samples are included.

Another table included is seen in Table 4.8 and shows the metrics for this model when the function "train-test-split" is used. This function split the dataset so 30% is used as testing data. The results show the confusion matrix and F1 score for the seven datasets as well for the time it takes to both train and test, which is over one hour for the larger datasets like dataset 4 (DS4).

4.3.3 KNN

The last model to include in this third experiment is the KNN model, it achieved an average F1-score of 93% in the first comparison found in Table 4.3. KNN does as the two other models contain a wide variety of options to choose from when attempting to optimize the model. One of the most attractive features here is to look at how many neighbors perform the best on this type of data. And to keep a manageable amount of testing no other options will be added as it would demand quite a bit of resources and time, as previously stated finding the best parameters for each model is out of scope as this serves as a foundation for further research. All problems are different and might require other combinations of parameters to perform optimally.

The default number of neighbors used in KNN is five, we tested the range between 1 and 40 neighbors on dataset 1. It would have been interesting to increase the range and see whether a higher number of neighbors are better in classifying this data, but due to restrictions in time and resources, it was decided that up to 40 would be sufficient for this thesis. A more profound and extensive experiment into optimizing each model would be recommended in future work.

Table 4.9: Time it takes for KNN to be trained and tested with the train-test-split function.

KNN	DS1	DS2	DS3	DS4	DS5	DS6	DS7
Accuracy	0.960	0.947	0.943	0.995	0.988	0.989	0.966
F1-score	0.932	0.948	0.911	0.872	0.922	0.907	0.877
TN	4232	1703	1798	89939	44881	89871	5453
FP	53	81	65	50	139	148	87
FN	186	111	91	181	407	812	126
TP	1654	1780	803	1479	3255	4695	766
Time	0:00:06	0:00:10	0:00:01	0:52:09	0:14:58	0:46:27	0:00:11

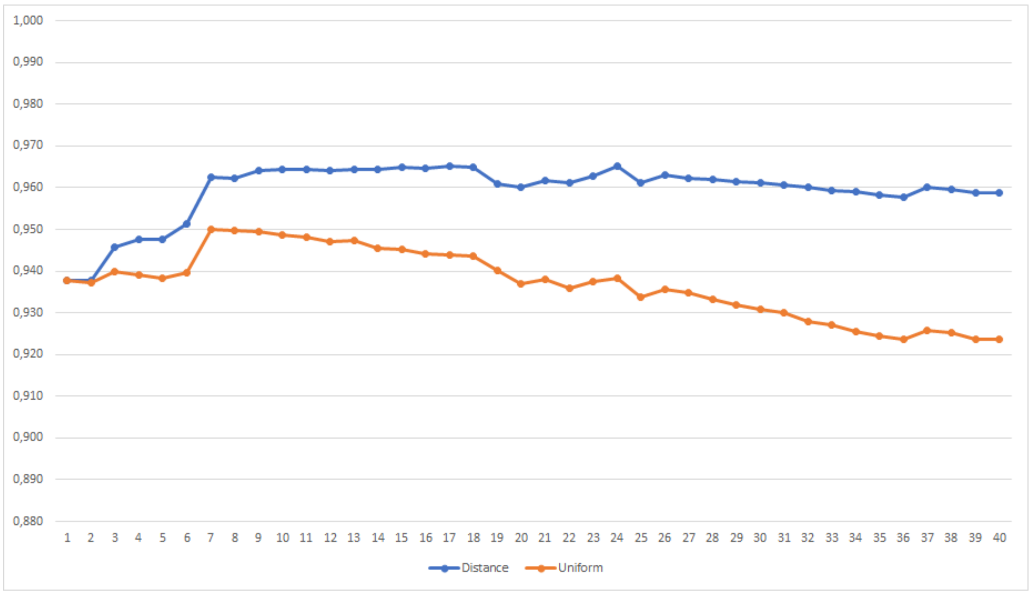


Figure 4.1: Performance with different number of neighbors.

As for KNN the results from testing the two distance measuring algorithms and the number of neighbors be seen in Figure 4.1. The graph shows that the default "distance" based option performs better than the "uniform" option. From the Figure 4.1 we can see that the score is quite high already at one neighbor with an F1-score right below 94%. It can then be seen that the performance increases and has a peak at seven neighbors. For the distance-based line can we see the performance remains the same and slightly increases between seven and 17 neighbors before it goes into a small drop. The line which is blue can then be seen maintaining a flat and slightly downward-facing line as the number of neighbors increases. The trend for the Uniform parameter can be seen decreasing slowly and steadily after the peak at 7 neighbors. The highest score is 96,5 and is found with 15,16,17 and 24 neighbors using the Distance-based parameter shown in blue.

Another table with results can also be seen in Table 4.9 and shows the time it takes to train and test the model using the "train-test-split" function. For each of the seven datasets are 30% used for testing and the rest for training, and the table shows the confusion matrix produced together with the F1 score and the time it takes to run. For KNN can we see that it takes 52 minutes to train and test the model on a larger dataset, dataset 4 (DS4)

4.3.4 Thresholds

The following subsection will present the results from looking into methods to decide on thresholds and further optimize the results of the models. As described in Section 3.3.3, the Precision-Recall (PR) curve will be used instead of the more common Receiver Operating (ROC) curve as most datasets are imbalanced, and more realistic results can be achieved with that method. When conducting these tests, each dataset was trained and tested with each of the three models with the "train-test-split" function. This is for reducing the number of results as cross-validation is looked at in other parts of the experiment. The training set equals 30% of the dataset while the remaining 70% is used for training. The models will be trained with the best options, as found in the previous section. Below will the results from each dataset be listed to view the trends. All three models are listed in each of the graphs, and the legend shows the word "AP" which stands for the Average Precision calculated from the graphs. Each graph is accompanied by a table showing the results from testing the models which were used to create the PR-curve, it consists of the precision and F1-score as well as the confusion matrix which consists of TN, FN, FP, and TP.

Table 4.10: Results from models making PR-curve for dataset 1

	MLPC	SVC	KNN
Precision	0.987	0.977	0.960
F1-score	0.978	0.962	0.932
TN - FP	4246 - 39	4248 - 37	4232 - 53
FN - TP	40 - 1800	99 - 1741	186 - 1654

Table 4.11: Results from models making PR-curve for dataset 2

	MLPC	SVC	KNN
Accuracy	0.982	0.974	0.947
F1-score	0.983	0.975	0.948
TN - FP	1748 - 36	1723 - 61	1703 - 81
FN - TP	27 - 1864	34 - 1857	111 - 1780

Table 4.12: Results from models making PR-curve for dataset 3

	MLPC	SVC	KNN
Accuracy	0.968	0.971	0.911
F1-score	0.968	0.955	0.911
TN - FP	1832 - 31	1848 - 15	1798 - 65
FN - TP	26 - 868	63 - 831	91 - 803

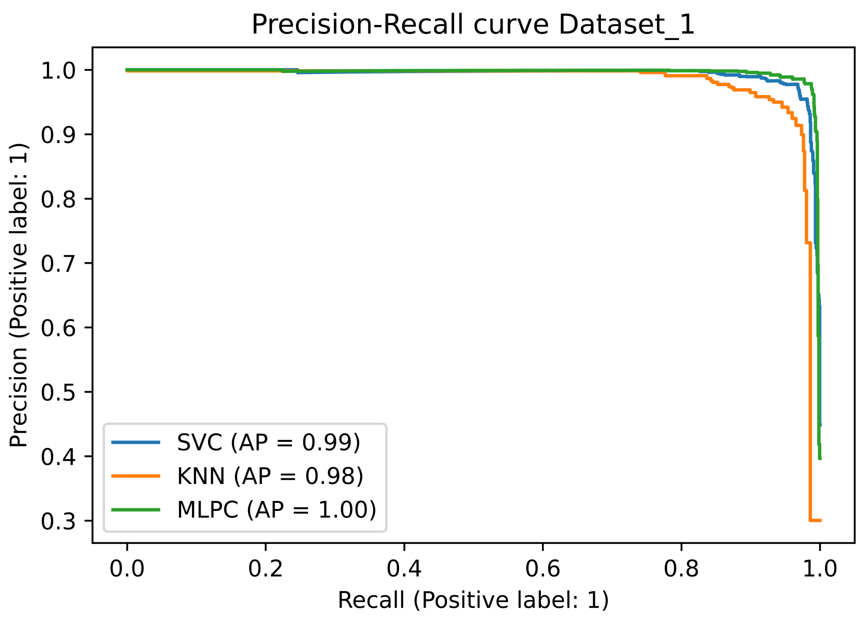


Figure 4.2: Precision-Recall curve for dataset 1.

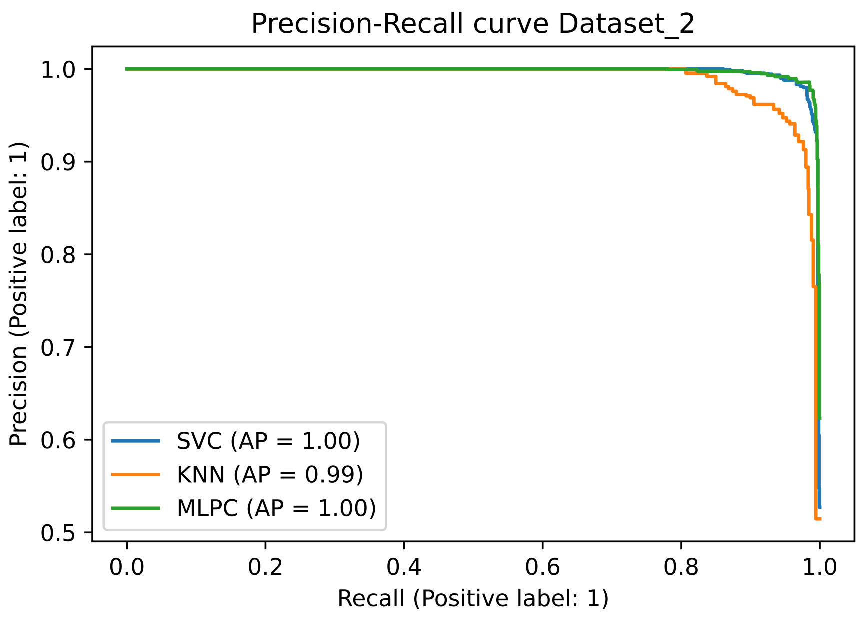


Figure 4.3: Precision-Recall curve for dataset 2.

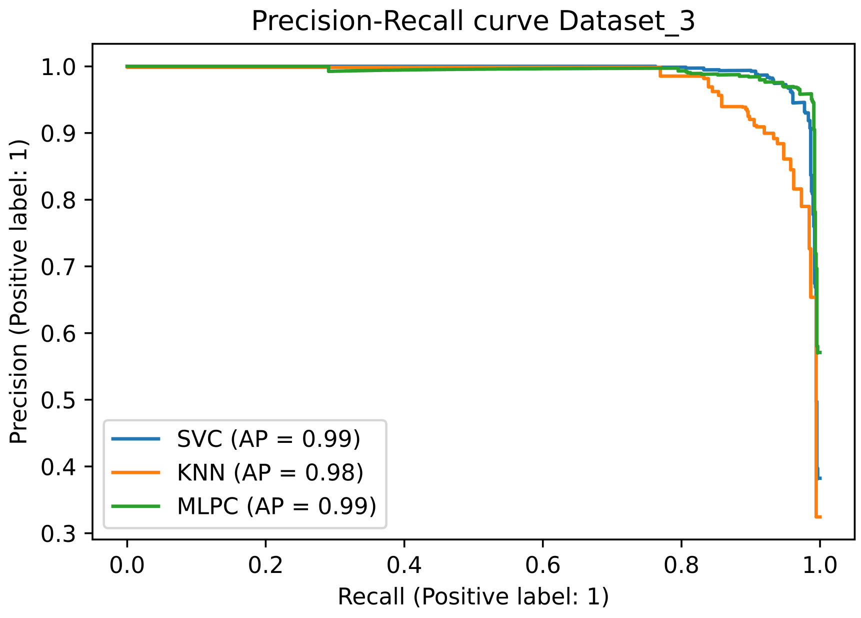


Figure 4.4: Precision-Recall curve for dataset 3.

Table 4.13: Results from models making PR-curve for dataset 4

	MLPC	SVC	KNN
Accuracy	0.996	0.995	0.995
F1-score	0.916	0.873	0.872
TN - FP FN - TP	88921 - 68 231 - 1631	89985 - 4 414 - 1448	89939 - 50 383 - 1479

Table 4.14: Results from models making PR-curve for dataset 5

	MLPC	SVC	KNN
Accuracy	0.992	0.989	0.988
F1-score	0.953	0.926	0.922
TN - FP FN - TP	44751 - 269 83 - 3579	44879 - 141 380 - 3282	44881 - 139 407 - 3255

Table 4.15: Results from models making PR-curve for dataset 6

	MLPC	SVC	KNN
Accuracy	0.993	0.992	0.989
F1-score	0.946	0.931	0.907
TN - FP FN - TP	89544 - 475 128 - 5379	89916 - 103 614 - 4893	89871 - 148 812 - 4695

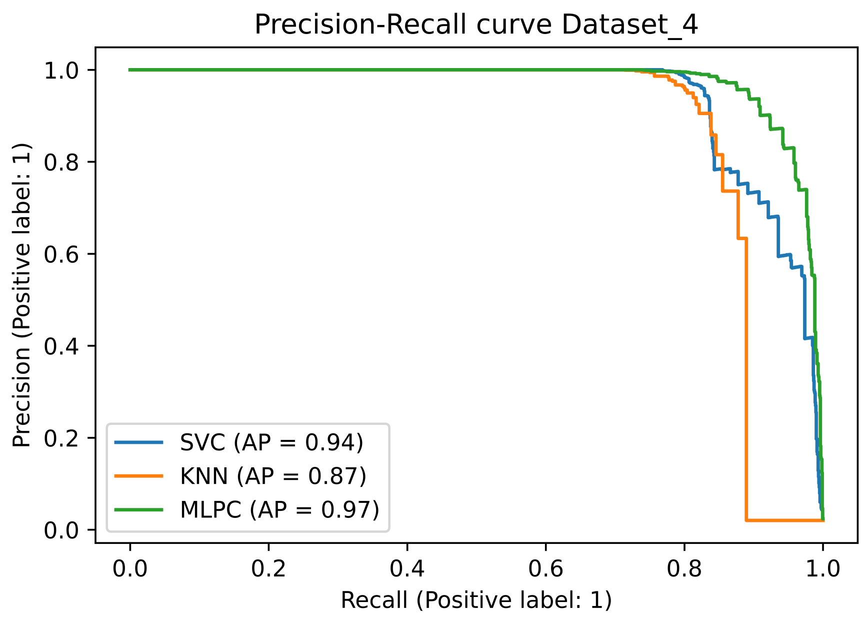


Figure 4.5: Precision-Recall curve for dataset 4.

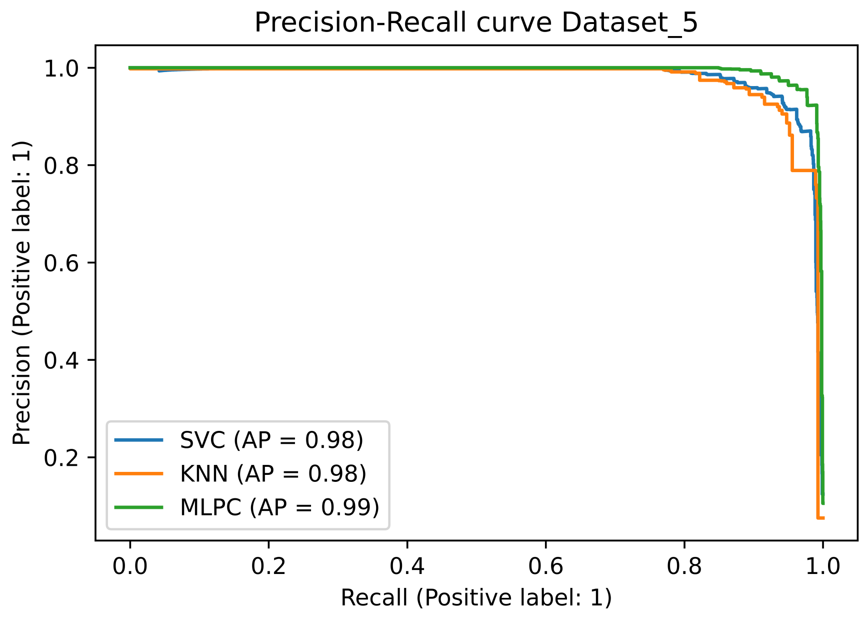


Figure 4.6: Precision-Recall curve for dataset 5.

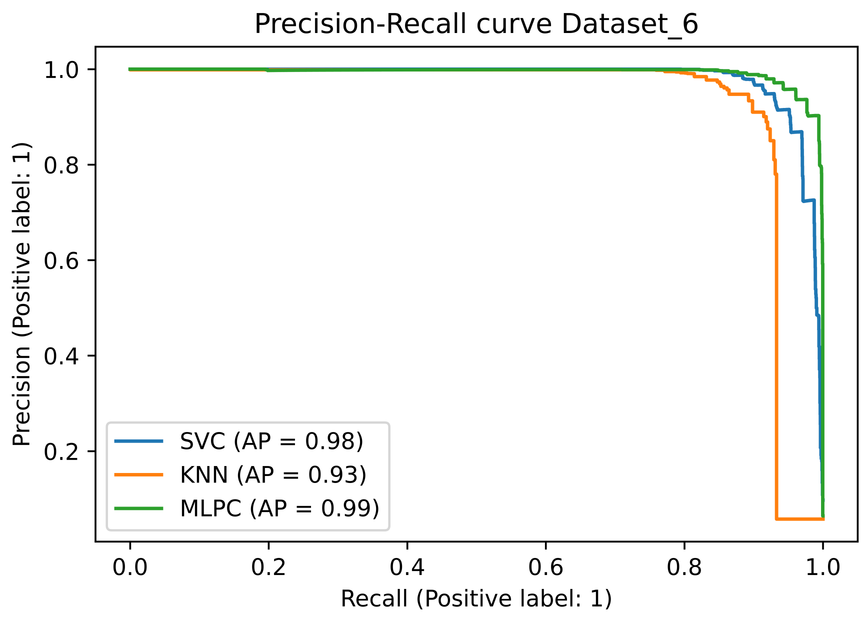


Figure 4.7: Precision-Recall curve for dataset 6.

Table 4.16: Results from models making PR-curve for dataset 7

	MLPC	SVC	KNN
Accuracy	0.988	0.980	0.966
F1-score	0.958	0.927	0.877
TN - FP FN - TP	5495 - 45 30 - 862	5503 - 37 88 - 804	5453 - 87 126 - 766

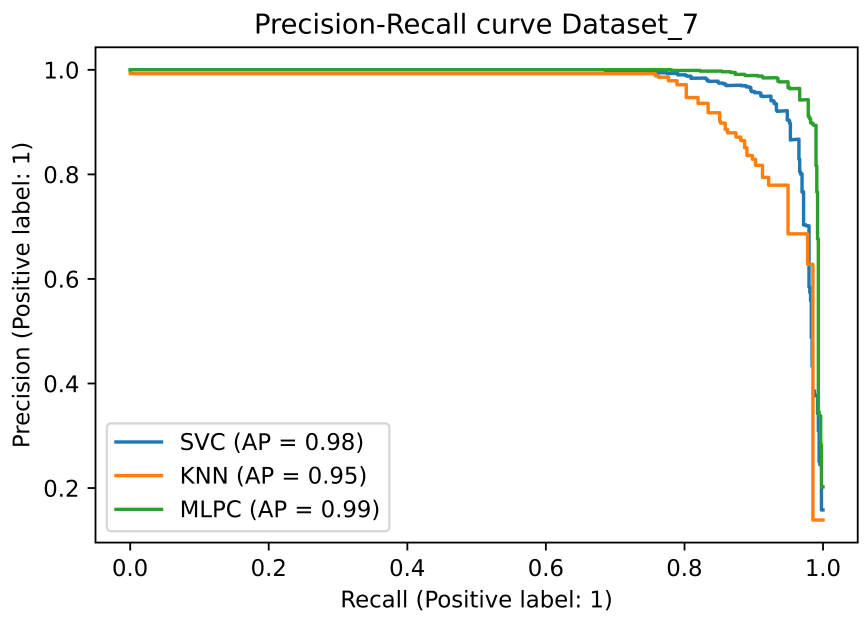


Figure 4.8: Precision-Recall curve for dataset 7.

From the results shown in the figures and tables above can some tendencies can be seen. Firstly we can see in all the figures that the orange line visualizing results from KNN performs a bit lower than the other two, similar to previously found results. For the first three datasets seen in Figure 4.2, Figure 4.3 and Figure 4.4 can we see quite high scores where the lines have a sharper curve up in the corner indicating both high precision and high recall.

However, when looking at the results from the graph in dataset 4 as seen in Figure 4.5 and the metrics in Table 4.13 can we see the scores are lower, and at least the line for KNN show that the precision drops faster when the recall increases even more than what is seen in the other graphs. The reason for bigger differences in this dataset might be the huge imbalance which is reduced in the others. This imbalance can make the model sacrifice the precision in order to increase the recall as there are so few occurrences of the minor class. This can be supported when looking at dataset 5 shown in Figure 4.6 where the balance of the dataset is better by removing some benign samples and duplicating the malicious ones. Here the results perform better and give higher scores for both precision and recall.

Confirming the results we already have achieved, MLPC performs overall better than both SVC and KNN. In some of the figures as seen above, they do perform quite similarly, but for the larger datasets, it is more evident that MLPC gives the best results with a curve closer to the upper right corner. However, one of the reasons for conducting this experiment is the need for the lowest amount of false positives to reduce the workload of analysts, as identified in related works and confirmed by experts. As we can see in the figures for dataset 4 Figure 4.5, dataset 5 Figure 4.6, and dataset 6 Figure 4.7, where all models already perform pretty well, is it possible to prioritize the precision and reducing the FP-rate by lowering the recall slightly. At least for MLPC which performs best, the loss of recall is low even when the precision is maintained.

Another observation from comparing the results found in dataset 4 Table 4.13,5 Table 4.14 and 6 Table 4.15 is the number of FP seems to be the lowest in dataset 4 where all the data used is compared to dataset 5 and 6 which uses the upsampling technique. The score increases, but with the increased number of FP, it might not be worth it based on these initial results.

4.4 Results experiment 4: Optimal number of features

In this next section will the three top models and the results from experiment 1: feature ranking be combined to find the optimal number of features. The results will be presented in the graph Figure 4.9 showing how the performance changes based on how many features are removed. A second table will also be used to present the exact results for a certain number of indicators. As seen in Table 4.17 are the F1-scores listed for each model in intervals of 50 features until 24 are left. The rest of the features and their scores are then shown, removing two at a time.

As this test is done on one of the smaller datasets it can be affected by the lack

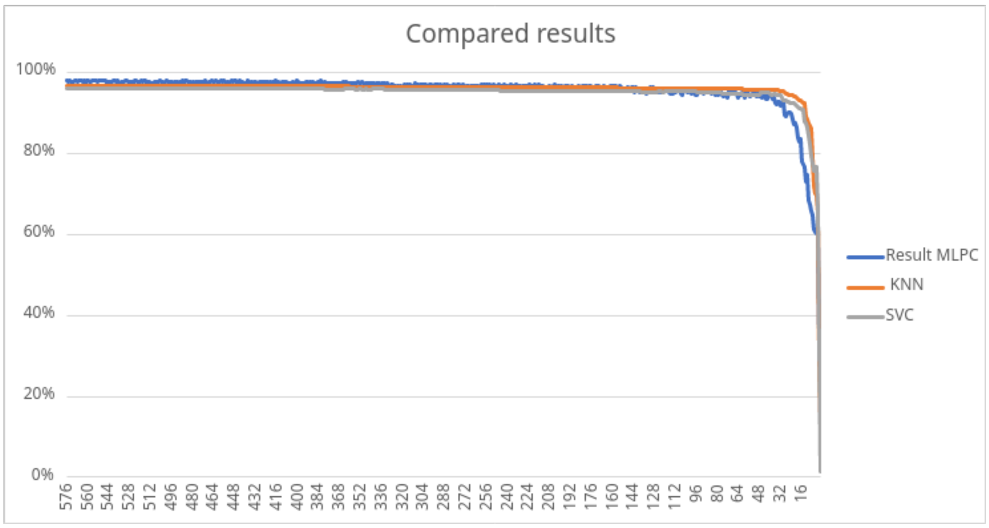


Figure 4.9: Figure comparing performance against the number of features.

of diversity of features and samples, but from the results in experiment 1, most of the top-rated features are the same. Several of the indicators used in this project are only found in a few of the samples. Since the occurrences of some indicators are pretty low, several of them can be removed without losing too much performance. The results from this experiment are shown in Table 4.17 and show that at least the first 200 features can be removed from MLPC and SVC with a minimal drop in performance. Looking closer at MLPC, we can see that the F1 score only drops from 97,9% to 97,3% when removing the 200 worst-ranked features as identified in experiment 1. This has minimal impact compared to the reduced amount of resources needed to compute the data, especially for larger datasets. If speed and resource usage is more important than the model's accuracy, even more indicators can be removed. With only 126 indicators the F1 score of MLPC will still stay at around 95,6% only reduced by 2,3% from using over 500 features.

The same tendencies can be seen across all three models, but another observation of the results from the KNN model is that it can keep a score of above 90% event when only relying on the top 14 features, the score is then still at 92%, higher than both of the two other models. What is interesting is that both SVC and KNN tend to perform better than MLPC when looking at using fewer features, but the highest scores are still found with MLPC using 176 features and higher. There could be several reasons for these results, and the most obvious one might be the optimization of models. Because the higher complexity of the MLPC model probably ruins some of the performance, at least for the lower number of indicators. We do still see that results achieved in experiment 1 do hold up as we can see relatively high scores even with a highly reduced number of features. Ideally, the same tests should be conducted for several of the datasets, mainly dataset 4, which contains all samples. But the time and resources needed to test and train the models so many times are more significant than what is seen as reasonable

Table 4.17: Overview selected number of features

Num indicators	MLPC	SVC	KNN
576	0,979	0,960	0,965
526	0,977	0,960	0,965
476	0,975	0,960	0,966
426	0,976	0,959	0,966
376	0,973	0,958	0,966
326	0,967	0,958	0,964
276	0,967	0,956	0,963
226	0,967	0,955	0,963
176	0,962	0,954	0,962
126	0,956	0,952	0,961
76	0,941	0,947	0,959
26	0,901	0,927	0,944
24	0,899	0,925	0,943
22	0,871	0,922	0,941
20	0,865	0,918	0,936
18	0,828	0,910	0,931
16	0,779	0,907	0,928
14	0,765	0,877	0,923
12	0,747	0,863	0,884
10	0,673	0,821	0,868
8	0,648	0,776	0,792
6	0,607	0,767	0,699
4	0,598	0,660	0,654
2	0,338	0,016	0,143

for this thesis and its research questions. It is therefore moved to future work to conduct a deeper analysis of the results and to expand on the experiments.

Another representation of the results can be seen in Figure 4.9, we can see that the performance of the models decreases quite slowly when removing features. First, when reaching around 40 features and below, we can see the curve go significantly downwards.

4.5 Results experiment 5: Model evaluation

In the following section will the presented results show how the models and data-sets performed against an untouched part of the data that was left out at the beginning of the project. This is called the verification data and has not been used in any of the experiments up until now, it consists of data from **75,028** benign and **1,514** malicious samples. The three top-performing models are trained with the best options identified in Section 4.3 on each of the initial datasets. This process is also done two times in order to find results from where all indicators are used and compare against only using the top 100. Based on the results in experiment 4 found in Section 4.4, the performance loss was low, and the impact will be tested in this part.

4.5.1 Re-test models

Table 4.18: Results from testing models against the designated test data.

Num features	SVC	KNN	MLPC
Dataset 1			
All	0,79	0,71	0,77
Top 100	0,77	0,64	0,74
Dataset 2			
All	0,64	0,56	0,68
Top 100	0,63	0,47	0,71
Dataset 3			
All	0,75	0,52	0,68
Top 100	0,74	0,50	0,68
Dataset 4			
All	0,88	0,86	0,91
Top 100	0,87	0,85	0,90
Dataset 5			
All	0,87	0,87	0,90
Top 100	0,86	0,84	0,89
Dataset 6			
All	0,90	0,87	**0,92**
Top 100	0,89	0,85	0,91
Dataset 7			
All	0,81	0,76	0,86
Top 100	0,80	0,76	0,83

The results from this experiment can be seen in Table 4.18, which shows the results for each model when trained on all indicators and the top 100. For each result entry, can the dataset used in the training process be seen, all seven were used. One general observation from these results is that the KNN model tends

Table 4.19: Extended Results for dataset 4, 5, and 6.

	Accuracy	F1-score	TN	FP	FN	TP
Dataset 4						
All features						
SVC	0,996	0,879	75025	3	324	1190
KNN	0,995	0,859	75006	22	357	1157
MLPC	0,997	0,910	74993	35	222	1292
Topp 100						
SVC	0,995	0,869	75025	3	349	1165
KNN	0,995	0,848	74994	34	374	1140
MLPC	0,996	0,901	75001	27	251	1263
Dataset 5						
All features						
SVC	0,995	0,868	74772	256	156	1358
KNN	0,995	0,866	74903	125	263	1251
MLPC	0,996	0,899	74813	215	102	1412
Topp 100						
SVC	0,994	0,861	74807	221	202	1312
KNN	0,994	0,839	74841	187	286	1228
MLPC	0,995	0,887	74807	221	130	1384
Dataset 6						
All features						
SVC	0,996	0,901	74905	123	171	1343
KNN	0,995	0,873	74931	97	266	1248
MLPC	0,997	0,924	74956	72	152	1362
Topp 100						
SVC	0,996	0,886	74890	138	201	1313
KNN	0,994	0,855	74883	145	276	1238
MLPC	0,997	0,912	74927	101	161	1353

to perform the worst among the three, while SVC and MLPC have more or less similar results. It seems to vary a bit between the datasets whether which one performs the best, such as in datasets 1 and 3, where SVC can be seen with higher scores than MLPC. It shows that by reducing the number of benign packages by downsampling, SVC might tend to perform better, but the overall highest results are seen from MLPC. The highest score can be seen marked in bold and is found in Dataset 6, which uses the upsampling technique to increase the number of malicious samples. Here are all the malicious packages duplicated two times, details can be seen in Table 3.2. From the results we have found during this experiment, there still seems to be the case that training with more data tends to improve the performance of the models. As the upsampling technique achieves the overall highest scores, not doing anything to the training data also achieved an F1 score of about 91%, which only differs 1% percent from the highest. The reason might be the type of samples and the number from the different categories. As the data have not been reviewed before the experiments, it is hard to decide precisely why the results ended up like this. Even though the primary training and testing data was split at the beginning of the project and an unfortunate split of features might have been the case since we don't know how the type of features are divided across the dataset other than the number of samples. If all samples of one malware category ended in only one of the datasets, it would be a natural reason why a particular part of the samples is misclassified.

When looking at the difference between using all the features or only the top 100 can wee see that the results only differ a few percentages in performance. Especially when looking at the results from dataset 4,5 and 6 where most of the data is used, does it only differ 1% percentage between all indicators compared to the top 100. That can be interpreted as most of the samples in the testing set are similar to what is found in the training data. It does not manage to classify everything but still provides relatively high scores compared to the amount of work that has been put into optimization. Looking closer at the results in the most realistic datasets, can we see some extended results in Table 4.19 where the number of correctly and wrongly classified samples are listed for the three datasets 4, 5, and 6. From what we can see in Table 4.19, the highest score and best compromise between false positive(FP) and false negative(FN) are using all features, dataset 6 for training and MLPC. With an F1-score of 92%, 72 wrongly classified benign and 152 malicious is the best outcome. The model still misses several samples, but compared to the almost 75,000 benign packages correctly classified, it can seem good at catching most attacks without too many FP. As this experiment was constructed and used to see what performance can be expected for the different models, we can see that with a larger dataset and upsampling it is possible to achieve a F1-score between 85% and 92% depending on the model and technique. Based on this experiment, the model to recommend is MLPC or similar variants, but SVC could also be of interest as it is not far behind MLPC in terms of the F1 scores found here. Both models also have many parameters for further optimization, possibly leading to even better results. Still, more care should also probably be given to the training data ensuring a good composition of different samples. Regarding how many features to use towards answering RQ2, utilizing only the top 100 features identified using the IG should also be considered. After removing over 400 features can we still see the models perform well, losing minimal performance. For some of the datasets, it only appears to differ 1%.

4.5.2 Statistics

The last subsection of this chapter will show some statistics from the results identified in experiment 5 and the features in the verification dataset. Three tables are present, the first being Table 4.20, which show the number of wrongly classified samples when using datasets 4,5, and 6 as the training data. All features and samples were used in the training data as it provided the best results. The table shows the total number of wrongly classified samples across all models, according to the category, for each model and the number of unique samples. This way, it is possible to see whether several of the misclassified samples overlap between the models.

The second Table 4.21 shows the top five features that occur among the malicious samples. The table also includes the number of appearances the indicator has in each category and the percentage. This will show to what degree the most common features are to appear in either malicious or benign packages.

The third and last Table 4.22 shows the average of the non-binary values comparing wrongly classified samples and the entire dataset. The average for the malicious samples and benign samples are also separated to see the average differences between the categories.

Table 4.20: Amount of wrongly classified samples using all features and the full training set.

	Total	Benign	Malicious	SVC	KNN	MLPC	Unique
Dataset 4	963	60	903	327	379	257	420
Dataset 5	1117	596	521	412	388	317	669
Dataset 6	881	292	589	294	363	224	489

Table 4.21: Test dataset statistics sorted by occurrences in malicious samples.

Indicator	Total	Benign	Malicious	Total%	Benign%	Malicious%
indicator27	48405	47058	1347	63.24	62.72	88.97
indicator77	13758	12699	1059	17.97	16.93	69.95
indicator80	11643	10599	1044	15.21	14.13	68.96
indicator4	1685	982	703	12.25	7.73	66.38
indicator22	44354	43808	546	57.95	58.39	36.06

The focus of in this part is to look closer at the statistics for dataset 4, 5, and 6 which is considered more realistic and contains the highest number of samples in training the models. First out are some statistics for how many samples are misclassified by each model on the three datasets. In Table 4.20 the number of wrongly classified samples for each model, the total in each dataset, and the number of unique samples are shown. The table shows that dataset 6 has the lowest overall number of misclassified samples when having the largest training set. However, the number of unique samples misclassified across all models is slightly higher than the highest number of misclassified samples in Dataset 4. SVC missed 327 samples, KNN 379, and MLPC 257, the number of unique missed samples is 420. This shows that most missed packages are the same across all the models, but some of the models miss a few other samples than the rest. The same tendencies can also be seen for datasets 5 and 6 as well where the number of unique samples is higher than for each model, this shows there are some commonalities among the models as to what type of samples is misclassified.

Looking away from the fact that three different models were used, one reason can be that duplicating the malicious samples seems to increase the number of FP by somewhat reducing the number of FN. Bringing back what was identified by Vu *et al.* [2], the need for a few FP as possible is necessary for the administrators as they have limited time and resources for dealing with wrongly classified samples. So based on the results found in this thesis can it seems using the full dataset for training without any modifications might be the best option, upsampling can however to a certain degree reduce the number of wrongly classified malicious samples, but seems to increase the number of wrongly classified benign.

Some general statistics can be seen in Table 4.21 which show the top 5 most common features among the malicious samples. The indicators are sorted by the number of occurrences found in the malicious samples and show the most common features among the malicious samples. Besides the numbers describing the occurrences is also the percentage for how often it occurs in its category compared to the total number of samples in the test dataset and the same for the number of benign or malicious. These results show that Indicator27 occurs in almost 89% of the malicious packages and right below 63% of benign. This indicator is triggered by the inclusion of the package "setuptools" which is commonly used in Python projects to include other files and handle external libraries. One reason it is less common among benign packages could be because many are made using standard functions and one or few files, contributing with trivial functionality which lowers the requirements of handling dependencies. Another reason could be several of the benign relies on the packaging option provided by PyPi itself. More interesting can be to look at the indicators 77, 80, and 4 which are found in between 66% and 69% malicious but only between 7% and 16% of benign. These indicators as described in Table 4.1 and relate to HTTP requests and can be used in many legitimate packages but are highly relevant in malicious use cases for reaching data on the internet. Either trying to download additional malware or reaching back to the attacker.

When looking at the highest-ranked feature across all datasets is the package size. It might seem unnatural, but when taking a look at the Table 4.21 can we see that the average package sizes for the missed samples are much higher than both the correctly classified and the full test set looking at each class individually. Several reasons could be the cause, but one that is more likely is the size and type of malicious packages that are found in the specific datasets used. If the models are trained on seeing small malicious samples in training, they may misclassify larger ones. This can be strengthened by looking at Table 4.22 which shows the average of non-binary features between the correctly and wrongly classified samples. Here can we see that the package sizes of the correctly classified samples are much lower compared to the samples wrongly classified. This could mean the larger samples are harder to classify and one reason for this might be the type of samples used in the training process. It is much more likely that a large number of the malicious samples seen are small and simple packages and with a higher number of larger and more complex malware samples, the results might be different. This could also be supported by looking at the average size of malicious samples in the verification dataset which is much lower compared to the wrongly classified. However, since the average is used in these statistics it might not be fully accurate, but since it is split up into both correctly and wrongly classified it will still give indications as to what is typical.

Table 4.22: Overview of average values for non-binary features in verification set, correctly and wrongly classified samples.

	Version	Package size	Py_files	Exec_files	Non_text_files	Num_indicators
Missed samples						
Mal	5,97	2210	6,34	1,17	1,48	7,73
Ben	7,52	5872	34,43	6,70	31,80	25,61
Correct samples						
Mal	2,52	887	9,62	0,81	2,22	13,02
Ben	4,69	929	16,65	0,52	44,15	15,63
Verification dataset						
Mal	3,36	1207	8,83	0,90	2,04	11,74
Ben	4,69	932	16,67	0,52	44,14	15,64

Chapter 5

Discussion and conclusion

The previous chapter shows all the results achieved during the experiments. The following chapter will discuss the discovered results and how they relate to the research questions. Afterward, will there be dedicated a section to describe the limitations of the thesis and then evaluate the validity of the thesis. Lastly, will the thesis be concluded.

5.1 Results

The results which were presented in the previous chapter will be discussed one research question at a time with what implications they might have.

5.1.1 RQ1

RQ1 was about looking into what indicator or combination of indicators are most commonly found among the malicious packages. This RQ was mainly answered in experiment 5 with the part on statistics. Several of the other experiments were also used to facilitate the last experiment and allowed us to not only look at what features are commonly found in the malicious packages but extend the question by also seeing what was typical for the missed samples. From the results of experiment 5, we could see that the indicator related to the use of the package "setuptool" was found in almost 89% of the malicious packages in the verification dataset. However, It was also quite common among benign packages and occurred in about 62% of the benign. The other indicators which were also commonly found among the malicious were related to the use of web GET requests, then a library for interacting with the operating system, and an unusual amount of string concatenations. Most of these indicators can often be seen in legitimate applications, but the unusual concatenation of strings stands out as potentially more malicious. This indicator was found in over 66% of the malicious and only 7% of the benign, meaning it could be better suited at identifying at least some of the malicious samples, which will be discussed more in the next section about RQ2. Further research is needed to identify what combinations of indicators are more common to appear among the malicious samples. That may require further analysis of the samples and their indicators, focusing closer on what combinations most often occur. It might even be necessary to divide the samples into their malware categories and find commonalities to answer that question further. Looking at single indicators that appear often is not enough by itself to indicate whether a sample is malicious or not since they use tools and techniques that very well could be used for legitimate purposes.

5.1.2 RQ2

RQ2 is also about indicators similar to RQ1 but aims to identify which features are better suited to classify benign and malicious PyPi packages. Several experiments have been contributing towards answering this RQ most noticeable and specific is experiment 1. The feature ranking was primarily directed towards answering both RQ1 and RQ2 but is also used as a foundation for some of the others. The indicators were only ranked with the information gain (IG) algorithm, and since there exist several different methods should several experiments be conducted to verify these results. However, we found from experiments 4 and 5 that only relying on the ranked indicators from IG provided nearly as high results as when all features were used. From experiment 4 seen in Section 4.4, did we find that we could utilize only the 26 highest ranked features and still preserve an F1 score over 90% for all three of the models, at least for testing on a subset of samples, in this case for dataset 1. This might change when using a more extensive and more diverse number of samples, but when relying on the top 100 indicators, can we from experiment 5 in Section 4.5 see that the models only lose 1% of the performance on the dataset 4, 5, and 6 compared to using all indicators. This also proves that by using the IG algorithm, we can remove many of the indicators to reduce the computational power and time needed to train and test the models with minimal loss of precision. Another key point from the performance of the 100 top features is that some of the over 500 indicators which have been used in this thesis are better than others. It is probably possible to narrow it down from the top 100 to even fewer with further research and experimentation to identify some specific indicator combinations that can correctly identify malicious and benign packages.

5.1.3 RQ3

For the last research question, RQ3, the focus was on identifying what performance could be expected from some of the most common Machine Learning (ML) models available. Experiments 2, 3, and 5 contributed to answering this RQ, testing a broader selection of models, followed by a low-effort attempt to optimize the top three before the results were eventually verified in the last experiment. Experiment 2 was the background for what models to use further in the thesis and resulted in MLPC, SVC, and KNN, which gave the best performance with the default settings and were the only models to have an average F1 score over 90%

across the seven datasets. Simple adjustments were tested on the models, and the general observation was that the default options performed among the best. KNN was tested differently by looking at how many neighbors performed the best, which showed that the numbers 15 to 17 are the best fit together with 7 and 24 in this use case and data.

Part of experiment 3 about optimizing the models was the precision-recall (PR) curve created for the seven datasets as shown in Section 4.3.4. Not directly answering the RQ but contributing towards an issue identified by Vu *et al.* [2] regarding the need for a low or close to zero false positive rate for the administrators to handle the workload. From the PR curves could we see that for the larger datasets 4, 5, and 6 MLPC was the best performing with SVC close after. It also seemed possible to improve precision meaning minimizing the number of false positives by reducing the threshold for the recall. How well it works and what performance to expect need to be examined further but it could be a viable path to follow to maximize the precision and reduce the number of FP.

To give a final answer to the RQ can the results found in experiment 5 that are written in Section 4.5 and table Table 4.19 show that we can expect an F1 score between 84% and 92% for the three used models with more extensive training sets. For others that want to look closer at ML and such static indicators is it a viable solution to invest a bit more into experimenting and optimizing the chosen model, we have at least contributed with a brief overview of what can be expected from SVC, KNN, and MLPC. All three do also have numerous options for further optimizing the models, and the results can probably be improved, but we achieved, at best, an F1 score of 92% with MLPC and upsampling the number of malicious samples in the dataset.

5.2 Limitations

There are some known limitations to the thesis and the achieved results. One general limitation of the thesis is the limited time frame of one semester. This resulted in a lower depth in some of the experiments than what would have been wanted. As some of the experiments were only conducted on single subsets of the full data, some results should also be verified with more extensive testing in future work with more time and resources. Some experiments can be quite time-consuming as seen in Section 4.3 and Table 4.6, over an hour is used to train and test MLPC on the larger datasets.

Another limitation is the dataset and the knowledge of what type of malicious packages are present. There have not been conducted deeper analyses or verification of the data other than it has been found in PyPi in the last four years. Initially, the datasets the data was divided into were also made using a pseudo-random function in Python with a given random state. As most results are based on these actions, some bias could be introduced by poorly dividing the samples and making them unbalanced regarding the composition of the samples. When picking what ML models to include in the thesis, did we rely on what was commonly found in the literature and what was documented on the internet so it would be easy to implement. The default options were also used for all models initially, which could be unfair to some models that could have performed better with other parameters. We did however decide on prioritizing the easiest approach and favorite the models which worked best "out-of-the-box"

Regarding the features, the version number was edited to easier fit the use of ML models. The version number was summed up and kept as a number instead of multiple digits which is normal. As other studies have looked into and tools are available to combat typosquatting and dependency confusion, was that aspect not prioritized.

5.3 Validity

Several aspects must be considered in terms of the validity of the thesis. One aspect is how the project was conducted since it was implemented in code with Python and the ML library SciKitLearn. We must acknowledge that mistakes could have occurred. The documentation and guides for using the tools have been followed but as known human error could still occur.

Another aspect of the validity of the results is the data used in the thesis. Based on the magnitude of the data it has not been possible to go through and verify what type of samples have been used, but as described in Section 3.2 does it contain most malicious samples that have been located in the PyPi repository in the later years. The number of malicious samples was also increased by using all known malicious versions of the same samples, which could make the achieved results biased toward some malware samples. As attack campaigns come in waves is there also likely to believe that several of the samples present stem from such a campaign and represents a part of the samples.

In regard to the experiments that have been conducted have the focus been on taking a broader look at features and ML model performance on static indicators. That means some of the experiments were only conducted on smaller datasets which might reflect only the data found in the specific set and not the whole data handled in the project or the real world.

5.4 Conclusion

In conclusion, this thesis aimed to look into what static indicators and features are commonly found in malicious Python packages, which are better for identifying these packages, and what performance could be expected from common machine learning models on this type of data. The motivation for conducting this project was to learn more about supply chain security in the Python Package Repository (PyPi) and contribute towards increasing the domain knowledge on how to combat malicious packages. From the cooperation with ReversingLabs, we were fortunate to have access to the static indicators and metadata of a larger dataset. Which was a great opportunity to provide more knowledge on the topic and show how well ML works together with this type of data.

We tried to answer these questions by conducting multiple experiments with common machine learning models and the feature ranking algorithm Information-Gain implemented from the Python library Scikitlearn. These experiments were then tested on multiple subsets of a dataset consisting of 375,073 benign packages and 7,639 malicious ones. From the results of training and testing a selection of model did we find that the Multi-Layer Perceptron (MLP) and Support Vector Machine (SVM) performs best when minimal work was put into optimization. MLP scored the highest when tested on the verification data, which were left out during the project, and had an F1 score of 92% using all features trained on a dataset where the number of malicious samples was increased with upsampling. When MLP was trained using the same data but only the top 100 indicators resulted in an F1 score of 91% and shows that over 400 indicators can be removed, and the performance only decreases by 1%. Without modifying the balance of the training data, the drop was only 1% which shows that a more balanced dataset might improve the accuracy of the models but, more importantly is a diverse and large training set. A brief experiment was also conducted, trying a few different parameters for the three top-performing models, which resulted in a minimal difference in results. However, this must be verified by more comprehensive experimentation.

Among the top indicators of malicious packages is the library for interacting with the web "Requests" and "Setuptools" for bundling libraries are most common. While the features about the package size, the number of indicators in a sample, the number of Python files, and indicator4, which triggers on unusual amount of string concatenations, are among the top-ranked features regarding how well they can divide the samples into categories. These results also show that there are many commonalities between both malicious and benign packages, but some features are still better suited. There is, however still a need to identify combinations of features that tend to indicate malicious activity over single indicators.

The thesis focused on the broader picture, and some of the experiments were conducted on a subset of samples, which might introduce bias into the results. The tools, models, and techniques are not too comprehensive so other alternatives could be better suited. More research is needed to optimize the models to increase performance. It is also recommended to dive deeper into the samples, especially the misclassified ones, and compare the attributes they have in common against those predicted correctly to see what the models miss and identify how they could be improved. Our contribution has improved domain knowledge in the field by showing what features were the highest ranked and proved more useful in classification based on a larger dataset of Python packages. We have also recommended two models that worked best out of the box for us during these experiments on this data, which could point to others where to continue with research into this topic. This information shows to some degree that static indicators work well to classify malicious samples with relatively high F1 scores with minimal effort and can be used as a supplement to discovering new threatening packages in an increasingly growing industry.

Chapter 6

Future work

In this short chapter will some of the items that have been mentioned as future work be presented. It will be discussed what could be done with the data before some ideas for new experiments and methods are listed.

6.1 Data

One of the issues which have been pointed out in previous research is the lack of data. This project has had access to quite a large dataset that spans multiple years but could as always probably contain more. However, one thing that would have been of high interest is a deeper analysis of what samples and types of malware actually are found in the dataset. As it could be quite time-consuming to go deeper into each of the samples, investigating and labeling all the data according to what malware type and family it belongs to and its capabilities. It would have been a great step for further increasing the knowledge about the dataset and would facilitate more and different experiments. It would also contribute to verifying whether how large part of the sample set belongs to attack campaigns or whether most samples are unique.

6.2 Methodology and experiments

From what has been done in this thesis regarding experiments several things could have been done in a different way and be extended upon. Especially when finding the best indicators to use, instead of only using the information gain (IG) algorithm, several different filters, wrappers, and embedded methods should be tested and compared. Both to see what features are prioritized across the different methods and to conduct further tests with the achieved results to find the best features and scoring methods. Another experiment related to the features which could be done much more extensively is finding the optimal number of features. In the thesis, only one downsampled dataset was used in this experiment, but with time and resources, it could be replicated to verify the results with other compositions of samples.

One method that could be used to look closer at the indicators and how each sample is classified is to utilize one of the tree algorithms. Even though they did not perform that well in these experiments, they have the ability to let the users easily follow the tree downwards to see what steps and decisions are taken and make it possible to debug where it misclassifies individual samples. Even though it might be demanding to follow large tree structures it that a possibility to get more familiar with how the model "thinks" about the data and to better understand how they could be improved in a more manual fashion.

The experiments conducted with machine learning (ML) models could have been extended in terms of what models to include in the initial comparison but also when it comes to optimizing each individually to see how high performance could be achieved. A few options were tested for only three of the models, the model implementations in Scikitlearn have more possibilities available for further optimizing the models. This is essential to look closer at if the optimal performance is wanted from the models, especially MLPC do probably have a huge potential in testing different numbers for the hidden layers parameter.

Another experiment that could have been interesting to test on the dataset is to use the unsupervised learning method clustering. To look further into what features tend to belong the use of clustering could potentially be helpful by discovering new patterns among the features and samples that have not been detected with the current methods.

Bibliography

[1] C. Flynn. 'Pypi stats - downlaod stats for downloads across all packages on pypi.' (), [Online]. Available: https://pypistats.org/packages/__all__ (visited on 26/04/2023).

[2] D.-L. Vu, Z. Newman and J. S. Meyers, 'A benchmark comparison of python malware detection approaches,' 2022. [Online]. Available: https://arxiv.org/abs/2209.13288 (visited on 27/10/2022).

[3] D. Ahmed, *Malicious pypi packages drop malware in new supply chain attack*. [Online]. Available: https://www.hackread.com/pypi-packages-malware-supply-chain-attack/ (visited on 15/03/2023).

[4] R. Jennings, *Pytorch supply chain attack: Dependency confusion burns devops*. [Online]. Available: https://www.reversinglabs.com/blog/pytorch-supply-chain-attack-dependency-confusion-burns-devops (visited on 15/03/2023).

[5] Wikipedia contributors, *Python (programming language) — Wikipedia, the free encyclopedia,* [Online; accessed 11-April-2023], 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Python_(programming_language)&oldid=1147632985.

[6] P. S. Foundation. 'Pypi statistics.' (), [Online]. Available: https://pypi.org/stats/ (visited on 26/04/2023).

[7] M. Sikorski, *Practical malware analysis : The hands-on guide to dissecting malicious software*, eng, San Francisco, 2012.

[8] C. Chio and D. Freeman, *Machine Learning and Security: Protecting Systems with Data and Algorithms*. "O'Reilly Media, Inc.", 26th Jan. 2018, 385 pp., ISBN: 978-1-4919-7987-7.

[9] M. Ohm, H. Plate, A. Sykosch and M. Meier, 'Backstabber's knife collection: A review of open source software supply chain attacks,' in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. Lecture Notes in Computer Science, vol. 12223, ISSN: 0302-9743, Cham: Springer International Publishing, 2020, pp. 23–43, ISBN: 978-3-030-52682-5. DOI: 10.1007/978-3-030-52683-2_2.

[10] E. Bommarito and M. Bommarito, *An empirical analysis of the python package index (PyPI)*, 25th Jul. 2019. DOI: 10.48550/arXiv.1907.11073. arXiv: 1907.11073[physics]. [Online]. Available: http://arxiv.org/abs/1907.11073 (visited on 01/11/2022).

[11] J. Ruohonen, K. Hjerppe and K. Rindell, 'A large-scale security-oriented static analysis of python packages in PyPI,' in *2021 18th International Conference on Privacy, Security and Trust (PST)*, Dec. 2021, pp. 1–10. DOI: 10.1109/PST52912.2021.9647791.

[12] B. Kaplan and J. Qian, 'A survey on common threats in npm and PyPi registries,' in *Communications in Computer and Information Science*, ser. Communications in Computer and Information Science, vol. 1482, ISSN: 1865-0929, Cham: Springer International Publishing, 2021, pp. 132–156, ISBN: 978-3-030-87838-2. DOI: 10.1007/978-3-030-87839-9_6.

[13] etal. 'CloudGuard spectral detects several malicious packages on PyPI - the official software repository for python developers,' Check Point Research. (8th Aug. 2022), [Online]. Available: https://research.checkpoint.com/2022/cloudguard-spectral-detects-several-malicious-packages-on-pypi-the-official-software-repository-for-python-developers/ (visited on 11/10/2022).

[14] P. Ladisa, H. Plate, M. Martinez and O. Barais, *Taxonomy of attacks on open-source software supply chains*, 19th Apr. 2022. DOI: 10.48550/arXiv.2204.04008. arXiv: 2204.04008[cs]. [Online]. Available: http://arxiv.org/abs/2204.04008 (visited on 09/01/2023).

[15] R. Duan, O. Alrawi, Ranjita Pai Kasturi, R. Elder, B. Saltaformaggio and W. Lee, 'Towards measuring supply chain attacks on package managers for interpreted languages,' *arXiv.org*, 2020, Place: Ithaca Publisher: Cornell University Library, arXiv.org, ISSN: 2331-8422.

[16] A. Zerouali, T. Mens, A. Decan and C. De Roover, 'On the impact of security vulnerabilities in the npm and RubyGems dependency networks,' *Empirical Software Engineering*, vol. 27, no. 5, p. 107, Sep. 2022, ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-022-10154-1. [Online]. Available: https://link.springer.com/10.1007/s10664-022-10154-1 (visited on 17/10/2022).

[17] M. Taylor, R. K. Vaidya, D. Davidson, L. De Carli and V. Rastogi, *SpellBound: Defending against package typosquatting*, 6th Mar. 2020. DOI: 10.48550/arXiv.2003.03471. arXiv: 2003.03471[cs]. [Online]. Available: http://arxiv.org/abs/2003.03471 (visited on 09/01/2023).

[18] D.-L. Vu, 'Py2src: Towards the automatic (and reliable) identification of sources for PyPI package,' Book Title: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Piscataway: IEEE, 2021, pp. 1394–1396, ISBN: 978-1-66540-337-5. DOI: 10.1109/ASE51524.

2021.9678526. [Online]. Available: https://ieeexplore.ieee.org/document/9678526 (visited on 17/10/2022).

[19] D. Ucci, L. Aniello and R. Baldoni, 'Survey of machine learning techniques for malware analysis,' *Computers & security*, vol. 81, pp. 123–147, 2019, Place: Amsterdam Publisher: Elsevier Ltd, ISSN: 0167-4048. DOI: 10.1016/j.cose.2018.11.001.

[20] A. Shalaginov, S. Banin, A. Dehghantanha and K. Franke, 'Machine learning aided static malware analysis: A survey and tutorial,' in vol. 70, 2018, pp. 7–45. DOI: 10.1007/978-3-319-73951-9_2. arXiv: 1808.01201[cs]. [Online]. Available: http://arxiv.org/abs/1808.01201 (visited on 09/01/2023).

[21] A. A. Milje, 'Detecting malicious python packages in the python package index (PyPI),' Accepted: 2022-08-04T17:19:20Z, Master thesis, NTNU, 2022. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3010208 (visited on 13/10/2022).

[22] Tensorflow, *Tensorflow*. [Online]. Available: https://www.tensorflow.org/ (visited on 08/05/2023).

[23] L. Foundation, *Pytourch*. [Online]. Available: https://pytorch.org/ (visited on 08/05/2023).

[24] S.-l. developers, *Sklearn.preprocessing.standardscaler*. [Online]. Available: https://scikit-learn.org/ (visited on 07/05/2023).

[25] P. Singh, S. K. Borgohain, L. D. Sharma and J. Kumar, 'Minimized feature overhead malware detection machine learning model employing MRMR-based ranking,' *Concurrency and computation*, vol. 34, no. 17, n/a, 2022, Place: Hoboken Publisher: Wiley Subscription Services, Inc, ISSN: 1532-0626. DOI: 10.1002/cpe.6992.

[26] ReversingLabs. 'Titanium platform.' (2023), [Online]. Available: https://www.reversinglabs.com/products/malware-analysis-platform.

[27] scikit-learn developers, *Sklearn.preprocessing.standardscaler*. [Online]. Available: https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler (visited on 07/05/2023).

[28] T. Saito and M. Rehmsmeier, 'The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets,' *PLoS ONE*, vol. 10, no. 3, e0118432, 4th Mar. 2015, ISSN: 1932-6203. DOI: 10.1371/journal.pone.0118432. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4349800/ (visited on 24/04/2023).

[29] S. Sharma, S. Sharma and A. Athaiya, 'ACTIVATION FUNCTIONS IN NEURAL NETWORKS,' *International Journal of Engineering Applied Sciences and Technology*, vol. 04, no. 12, pp. 310–316, 10th May 2020, ISSN: 24552143. DOI: `10.33564/IJEAST.2020.v04i12.054`. [Online]. Available: `https://www.ijeast.com/papers/310-316,Tesma412,IJEAST.pdf` (visited on 24/04/2023).

[30] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 29th Jan. 2017. DOI: `10.48550/arXiv.1412.6980`. arXiv: `1412.6980[cs]`. [Online]. Available: `http://arxiv.org/abs/1412.6980` (visited on 24/04/2023).

[31] scikit-learn developers, *Sklearn.neural*$_n$$etwork.mlpclassifier$. [Online]. Available: `https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html` (visited on 07/05/2023).

[32] scikit-learn developers, *Support vector machines*. [Online]. Available: `https://scikit-learn.org/stable/modules/svm.html#svc` (visited on 07/05/2023).

Appendix A

Experiment 2: Model comparison

A.1 5-fold cross-validation

fold	accuracy	f1-score	TP	FN	FP	TN
Dataset_1	0,00000	0,00000	0.0	0.0	0.0	0.0
RandomForestClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,79505	0,49057	2844.0	0.0	837.0	403.0
2.0	0,80480	0,50281	2883.0	0.0	797.0	403.0
3.0	0,98653	0,97761	2827.0	13.0	42.0	1201.0
4.0	0,95420	0,91859	2841.0	2.0	185.0	1055.0
5.0	0,94783	0,90427	2864.0	0.0	213.0	1006.0
0.7587694758445607						
DecisionTreeClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,73433	0,23104	2836.0	8.0	1077.0	163.0
2.0	0,67818	0,63581	1622.0	1261.0	53.0	1147.0
3.0	0,59270	0,59029	1222.0	1618.0	45.0	1198.0
4.0	0,60568	0,59649	1283.0	1560.0	50.0	1190.0
5.0	0,72422	0,25823	2761.0	103.0	1023.0	196.0
0.4623740099575916						
SVC	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,97453	0,95720	2817.0	27.0	77.0	1163.0
2.0	0,97526	0,95671	2866.0	17.0	84.0	1116.0
3.0	0,97771	0,96251	2824.0	16.0	75.0	1168.0
4.0	0,97820	0,96342	2822.0	21.0	68.0	1172.0
5.0	0,97649	0,95997	2836.0	28.0	68.0	1151.0
0.9599602366063833						
GaussianNB	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,57958	0,58953	1134.0	1710.0	7.0	1233.0
2.0	0,56356	0,57328	1104.0	1779.0	3.0	1197.0
3.0	0,58413	0,59241	1151.0	1689.0	9.0	1234.0
4.0	0,58780	0,59397	1169.0	1674.0	9.0	1231.0
5.0	0,58339	0,58804	1168.0	1696.0	5.0	1214.0
0.5874448351031785						
KNeighborsClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,95593	0,92208	2839.0	5.0	175.0	1065.0
2.0	0,96081	0,92908	2875.0	8.0	152.0	1048.0
3.0	0,96890	0,94693	2823.0	17.0	110.0	1133.0
4.0	0,98335	0,97218	2827.0	16.0	52.0	1188.0
5.0	0,95347	0,91630	2853.0	11.0	179.0	1040.0
0.9373121593448188						
GradientBoostingClassifie	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,72233	0,15875	2843.0	1.0	1133.0	107.0
2.0	0,76414	0,44560	2733.0	150.0	813.0	387.0
3.0	0,78619	0,73229	2016.0	824.0	49.0	1194.0
4.0	0,85329	0,80093	2279.0	564.0	35.0	1205.0
5.0	0,79500	0,49426	2837.0	27.0	810.0	409.0
0.526366126586417						
MLPClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,98310	0,97252	2794.0	50.0	19.0	1221.0
2.0	0,98775	0,97918	2857.0	26.0	24.0	1176.0
3.0	0,98849	0,98124	2807.0	33.0	14.0	1229.0
4.0	0,99143	0,98601	2815.0	28.0	7.0	1233.0
5.0	0,98702	0,97860	2818.0	46.0	7.0	1212.0
0.979510238601707						
fold	accuracy	f1-score	TP	FN	FP	TN
Dataset_2	0,00000	0,00000	0.0	0.0	0.0	0.0

RandomForestClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,96898	0,96880	1194.0	61.0	15.0	1180.0
2.0	0,63429	0,46539	1164.0	0.0	896.0	390.0
3.0	0,68490	0,53772	1229.0	0.0	772.0	449.0
4.0	0,91592	0,90645	1246.0	0.0	206.0	998.0
5.0	0,96694	0,96807	1141.0	51.0	30.0	1228.0
0.7692881737796351						
DecisionTreeClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,79551	0,80783	896.0	359.0	142.0	1053.0
2.0	0,53306	0,20334	1160.0	4.0	1140.0	146.0
3.0	0,79878	0,76580	1151.0	78.0	415.0	806.0
4.0	0,63918	0,72025	428.0	818.0	66.0	1138.0
5.0	0,74531	0,79269	633.0	559.0	65.0	1193.0
0.6579815247226706						
SVC	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,96776	0,96726	1204.0	51.0	28.0	1167.0
2.0	0,97714	0,97824	1135.0	29.0	27.0	1259.0
3.0	0,97429	0,97442	1187.0	42.0	21.0	1200.0
4.0	0,97061	0,97032	1201.0	45.0	27.0	1177.0
5.0	0,96735	0,96873	1131.0	61.0	19.0	1239.0
0.9717946341256789						
GaussianNB	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,72082	0,77603	581.0	674.0	10.0	1185.0
2.0	0,71102	0,78388	458.0	706.0	2.0	1284.0
3.0	0,69143	0,76226	482.0	747.0	9.0	1212.0
4.0	0,68694	0,75720	487.0	759.0	8.0	1196.0
5.0	0,70898	0,77823	486.0	706.0	7.0	1251.0
0.7715214151509158						
KNeighborsClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,96816	0,96755	1209.0	46.0	32.0	1163.0
2.0	0,93102	0,93014	1156.0	8.0	161.0	1125.0
3.0	0,95469	0,95267	1222.0	7.0	104.0	1117.0
4.0	0,94367	0,94000	1231.0	15.0	123.0	1081.0
5.0	0,97184	0,97230	1170.0	22.0	47.0	1211.0
0.952531204552314						
GradientBoostingClassifie	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,72204	0,77796	576.0	679.0	2.0	1193.0
2.0	0,61143	0,41451	1161.0	3.0	949.0	337.0
3.0	0,66163	0,50090	1205.0	24.0	805.0	416.0
4.0	0,64531	0,48181	1177.0	69.0	800.0	404.0
5.0	0,73469	0,79417	546.0	646.0	4.0	1254.0
0.5938724843043044						
MLPClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,97755	0,97726	1213.0	42.0	13.0	1182.0
2.0	0,98449	0,98532	1137.0	27.0	11.0	1275.0
3.0	0,98490	0,98485	1210.0	19.0	18.0	1203.0
4.0	0,98612	0,98594	1224.0	22.0	12.0	1192.0
5.0	0,97306	0,97426	1135.0	57.0	9.0	1249.0
0.981526525710216						
fold	accuracy	f1-score	TP	FN	FP	TN
Dataset_3	0,00000	0,00000	0.0	0.0	0.0	0.0
RandomForestClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,95049	0,92025	1222.0	0.0	91.0	525.0

2.0	0,78455	0,50868	1237.0	0.0	396.0	205.0
3.0	0,78280	0,53442	1209.0	0.0	399.0	229.0
4.0	0,97278	0,95994	1188.0	12.0	38.0	599.0
5.0	0,98530	0,97634	1253.0	5.0	22.0	557.0
0.7799250220710253						
DecisionTreeClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,70294	0,50364	1015.0	207.0	339.0	277.0
2.0	0,71273	0,23256	1230.0	7.0	521.0	80.0
3.0	0,74905	0,43436	1199.0	10.0	451.0	177.0
4.0	0,46652	0,56367	224.0	976.0	4.0	633.0
5.0	0,85955	0,76964	1148.0	110.0	148.0	431.0
0.5007723865939028						
SVC	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,97987	0,96985	1206.0	16.0	21.0	595.0
2.0	0,97443	0,95973	1231.0	6.0	41.0	560.0
3.0	0,96843	0,95307	1190.0	19.0	39.0	589.0
4.0	0,97115	0,95709	1193.0	7.0	46.0	591.0
5.0	0,97550	0,96056	1244.0	14.0	31.0	548.0
0.9600582617571906						
GaussianNB	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,55060	0,59825	397.0	825.0	1.0	615.0
2.0	0,57726	0,60498	466.0	771.0	6.0	595.0
3.0	0,55308	0,60126	397.0	812.0	9.0	619.0
4.0	0,54709	0,60343	372.0	828.0	4.0	633.0
5.0	0,52695	0,57044	391.0	867.0	2.0	577.0
0.5956731526159827						
KNeighborsClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,96899	0,95230	1212.0	10.0	47.0	569.0
2.0	0,96899	0,95065	1232.0	5.0	52.0	549.0
3.0	0,96788	0,95136	1201.0	8.0	51.0	577.0
4.0	0,95101	0,92500	1192.0	8.0	82.0	555.0
5.0	0,97333	0,95690	1244.0	14.0	35.0	544.0
0.9472430006746702						
GradientBoostingClassifie	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,76768	0,50058	1197.0	25.0	402.0	214.0
2.0	0,71219	0,22091	1234.0	3.0	526.0	75.0
3.0	0,70169	0,23249	1206.0	3.0	545.0	83.0
4.0	0,57267	0,61763	418.0	782.0	3.0	634.0
5.0	0,56287	0,59010	456.0	802.0	1.0	578.0
0.43234412417546747						
MLPClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,98640	0,97998	1201.0	21.0	4.0	612.0
2.0	0,98368	0,97496	1224.0	13.0	17.0	584.0
3.0	0,98204	0,97408	1184.0	25.0	8.0	620.0
4.0	0,95700	0,93881	1152.0	48.0	31.0	606.0
5.0	0,97224	0,95732	1214.0	44.0	7.0	572.0
0.9650297072914347						
fold	accuracy	f1-score	TP	FN	FP	TN
Dataset_4	0,00000	0,00000	0.0	0.0	0.0	0.0
RandomForestClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,99579	0,88111	60020.0	0.0	258.0	956.0
2.0	0,99562	0,88438	59941.0	0.0	268.0	1025.0
3.0	0,98602	0,44416	60036.0	2.0	854.0	342.0

4.0	0,98553	0,45443	59979.0	6.0	880.0	369.0
5.0	0,99536	0,86840	60013.0	3.0	281.0	937.0
0.7064950169846604						
DecisionTreeClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,97080	0,13035	59312.0	708.0	1080.0	134.0
2.0	0,97269	0,10397	59465.0	476.0	1196.0	97.0
3.0	0,76823	0,09582	46290.0	13748.0	444.0	752.0
4.0	0,67745	0,08641	40549.0	19436.0	315.0	934.0
5.0	0,83973	0,14631	50579.0	9437.0	377.0	841.0
0.11257074501976769						
SVC	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,99615	0,89263	60017.0	3.0	233.0	981.0
2.0	0,99546	0,87997	59937.0	4.0	274.0	1019.0
3.0	0,99572	0,87757	60033.0	5.0	257.0	939.0
4.0	0,99561	0,87943	59984.0	1.0	268.0	981.0
5.0	0,99551	0,87298	60014.0	2.0	273.0	945.0
0.8805141391005658						
GaussianNB	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,37618	0,05949	21827.0	38193.0	6.0	1208.0
2.0	0,37448	0,06279	21648.0	38293.0	10.0	1283.0
3.0	0,36372	0,05757	21082.0	38956.0	6.0	1190.0
4.0	0,38083	0,06153	22077.0	37908.0	6.0	1243.0
5.0	0,36904	0,05904	21386.0	38630.0	6.0	1212.0
0.06008203440205334						
KNeighborsClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,99641	0,90090	60014.0	6.0	214.0	1000.0
2.0	0,99562	0,88605	59924.0	17.0	251.0	1042.0
3.0	0,99610	0,89062	60022.0	16.0	223.0	973.0
4.0	0,99637	0,90406	59966.0	19.0	203.0	1046.0
5.0	0,99642	0,90210	60006.0	10.0	209.0	1009.0
0.8967472859415009						
GradientBoostingClassifie	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,98411	0,39149	59948.0	72.0	901.0	313.0
2.0	0,98310	0,38430	59876.0	65.0	970.0	323.0
3.0	0,93801	0,28512	56681.0	3357.0	439.0	757.0
4.0	0,92137	0,27211	55519.0	4466.0	349.0	900.0
5.0	0,93326	0,28959	56314.0	3702.0	385.0	833.0
0.3245218083492121						
MLPClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,99668	0,91636	59919.0	101.0	102.0	1112.0
2.0	0,99610	0,90902	59801.0	140.0	99.0	1194.0
3.0	0,99675	0,91120	60014.0	24.0	175.0	1021.0
4.0	0,99657	0,90987	59964.0	21.0	189.0	1060.0
5.0	0,99686	0,91767	59972.0	44.0	148.0	1070.0
0.9128236361951224						

fold	accuracy	f1-score	TP	FN	FP	TN
Dataset_5	0,00000	0,00000	0.0	0.0	0.0	0.0
RandomForestClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,98496	0,89604	29864.0	35.0	453.0	2103.0
2.0	0,95067	0,51055	30019.0	1.0	1600.0	835.0
3.0	0,98706	0,91029	29903.0	62.0	358.0	2131.0
4.0	0,98681	0,90585	29967.0	2.0	426.0	2059.0
5.0	0,94981	0,49078	30040.0	1.0	1628.0	785.0

0.7427018279710997						
DecisionTreeClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,92534	0,39980	29225.0	674.0	1749.0	807.0
2.0	0,97258	0,80525	29725.0	295.0	595.0	1840.0
3.0	0,76320	0,07398	24462.0	5503.0	2182.0	307.0
4.0	0,94115	0,66503	28648.0	1321.0	589.0	1896.0
5.0	0,87031	0,08837	28041.0	2000.0	2209.0	204.0
0.40648743531112147						
SVC	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,98955	0,93216	29787.0	112.0	227.0	2329.0
2.0	0,98937	0,92698	29920.0	100.0	245.0	2190.0
3.0	0,98971	0,93156	29847.0	118.0	216.0	2273.0
4.0	0,99032	0,93563	29858.0	111.0	203.0	2282.0
5.0	0,98845	0,91821	29974.0	67.0	308.0	2105.0
0.92890834306743						
GaussianNB	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,43768	0,21775	11665.0	18234.0	16.0	2540.0
2.0	0,42086	0,20471	11240.0	18780.0	16.0	2419.0
3.0	0,43267	0,21215	11563.0	18402.0	10.0	2479.0
4.0	0,43175	0,21181	11534.0	18435.0	7.0	2478.0
5.0	0,43492	0,20724	11718.0	18323.0	16.0	2397.0
0.21073044022400503						
KNeighborsClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,98721	0,91209	29887.0	12.0	403.0	2153.0
2.0	0,98749	0,91022	29991.0	29.0	377.0	2058.0
3.0	0,98955	0,92816	29925.0	40.0	299.0	2190.0
4.0	0,98746	0,91169	29946.0	23.0	384.0	2101.0
5.0	0,98835	0,91563	30025.0	16.0	362.0	2051.0
0.9155587741000015						
GradientBoostingClassifie	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,94300	0,46870	29789.0	110.0	1740.0	816.0
2.0	0,94494	0,42998	29994.0	26.0	1761.0	674.0
3.0	0,88334	0,52052	26613.0	3352.0	434.0	2055.0
4.0	0,94503	0,45939	29912.0	57.0	1727.0	758.0
5.0	0,90923	0,28426	28923.0	1118.0	1828.0	585.0
0.43256948383192084						
MLPClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,99242	0,95264	29735.0	164.0	82.0	2474.0
2.0	0,99150	0,94080	29986.0	34.0	242.0	2193.0
3.0	0,99106	0,94453	29695.0	270.0	20.0	2469.0
4.0	0,99267	0,95339	29782.0	187.0	51.0	2434.0
5.0	0,98857	0,91724	30027.0	14.0	357.0	2056.0
0.9417192289749629						
fold	accuracy	f1-score	TP	FN	FP	TN
Dataset_6	0,00000	0,00000	0.0	0.0	0.0	0.0
RandomForestClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,96148	0,49661	60021.0	4.0	2449.0	1210.0
2.0	0,98913	0,89965	59890.0	117.0	575.0	3102.0
3.0	0,95724	0,43495	59913.0	2.0	2721.0	1048.0
4.0	0,96057	0,47192	60051.0	1.0	2510.0	1122.0
5.0	0,98957	0,90616	59814.0	114.0	550.0	3206.0
0.6418593880927375						
DecisionTreeClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0

1.0	0,93692	0,36248	58525.0	1500.0	2517.0	1142.0
2.0	0,97469	0,77328	59323.0	684.0	928.0	2749.0
3.0	0,94691	0,19634	59890.0	25.0	3356.0	413.0
4.0	0,94755	0,15486	60038.0	14.0	3326.0	306.0
5.0	0,74328	0,27276	44269.0	15659.0	690.0	3066.0
0.351944124240413						
SVC	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,99220	0,92881	59945.0	80.0	417.0	3242.0
2.0	0,99284	0,93515	59940.0	67.0	389.0	3288.0
3.0	0,99165	0,92536	59854.0	61.0	471.0	3298.0
4.0	0,99267	0,93252	59990.0	62.0	405.0	3227.0
5.0	0,99267	0,93535	59839.0	89.0	378.0	3378.0
0.9314389459137418						
GaussianNB	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,38386	0,15646	20807.0	39218.0	20.0	3639.0
2.0	0,42147	0,16564	23184.0	36823.0	20.0	3657.0
3.0	0,38322	0,16015	20660.0	39255.0	24.0	3745.0
4.0	0,42529	0,16492	23470.0	36582.0	18.0	3614.0
5.0	0,38887	0,16092	21033.0	38895.0	24.0	3732.0
0.1616171586061061						
KNeighborsClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,99125	0,91870	59980.0	45.0	512.0	3147.0
2.0	0,99085	0,91473	59974.0	33.0	550.0	3127.0
3.0	0,99058	0,91404	59894.0	21.0	579.0	3190.0
4.0	0,99132	0,91813	60030.0	22.0	531.0	3101.0
5.0	0,99089	0,91705	59898.0	30.0	550.0	3206.0
0.9165299134042932						
GradientBoostingClassifie	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,95707	0,43254	59908.0	117.0	2617.0	1042.0
2.0	0,96008	0,69454	58252.0	1755.0	787.0	2890.0
3.0	0,94438	0,11627	59909.0	6.0	3536.0	233.0
4.0	0,94565	0,09184	60048.0	4.0	3457.0	175.0
5.0	0,92629	0,56721	55914.0	4014.0	680.0	3076.0
0.3804819603192523						
MLPClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,99044	0,92273	59439.0	586.0	23.0	3636.0
2.0	0,99422	0,95134	59719.0	288.0	80.0	3597.0
3.0	0,99256	0,93359	59878.0	37.0	437.0	3332.0
4.0	0,99375	0,94250	60024.0	28.0	370.0	3262.0
5.0	0,99256	0,94015	59487.0	441.0	33.0	3723.0
0.938061938917856						
fold	accuracy	f1-score	TP	FN	FP	TN
Dataset_7	0,00000	0,00000	0.0	0.0	0.0	0.0
RandomForestClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,97411	0,90133	3670.0	0.0	111.0	507.0
2.0	0,98507	0,94693	3653.0	12.0	52.0	571.0
3.0	0,97341	0,89425	3691.0	2.0	112.0	482.0
4.0	0,97317	0,89342	3690.0	0.0	115.0	482.0
5.0	0,90203	0,48403	3670.0	0.0	420.0	197.0
0.8239926531419016						
DecisionTreeClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,87570	0,41364	3567.0	103.0	430.0	188.0
2.0	0,73484	0,49668	2590.0	1075.0	62.0	561.0

3.0	0,79496	0,55130	2868.0	825.0	54.0	540.0
4.0	0,86914	0,62018	3268.0	422.0	139.0	458.0
5.0	0,86214	0,10860	3660.0	10.0	581.0	36.0
0.4380792626229339						
SVC	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,98274	0,93874	3647.0	23.0	51.0	567.0
2.0	0,98134	0,93421	3640.0	25.0	55.0	568.0
3.0	0,98041	0,92606	3677.0	16.0	68.0	526.0
4.0	0,98460	0,94291	3676.0	14.0	52.0	545.0
5.0	0,98530	0,94737	3657.0	13.0	50.0	567.0
0.9378567163290716						
GaussianNB	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,48624	0,35903	1468.0	2202.0	1.0	617.0
2.0	0,47808	0,35727	1428.0	2237.0	1.0	622.0
3.0	0,48542	0,34965	1488.0	2205.0	1.0	593.0
4.0	0,49895	0,35534	1547.0	2143.0	5.0	592.0
5.0	0,49009	0,35932	1488.0	2182.0	4.0	613.0
0.35612167811874107						
KNeighborsClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,97808	0,91840	3665.0	5.0	89.0	529.0
2.0	0,97948	0,92542	3654.0	11.0	77.0	546.0
3.0	0,98437	0,94199	3676.0	17.0	50.0	544.0
4.0	0,98577	0,94728	3678.0	12.0	49.0	548.0
5.0	0,98064	0,92801	3669.0	1.0	82.0	535.0
0.9322218334170322						
GradientBoostingClassifie	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,88969	0,41388	3648.0	22.0	451.0	167.0
2.0	0,82673	0,61201	2959.0	706.0	37.0	586.0
3.0	0,80453	0,56625	2902.0	791.0	47.0	547.0
4.0	0,86331	0,62242	3218.0	472.0	114.0	483.0
5.0	0,86121	0,06886	3670.0	0.0	595.0	22.0
0.4566843729660235						
MLPClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,98787	0,95847	3636.0	34.0	18.0	600.0
2.0	0,98811	0,95975	3629.0	36.0	15.0	608.0
3.0	0,98227	0,93851	3631.0	62.0	14.0	580.0
4.0	0,98787	0,95667	3661.0	29.0	23.0	574.0
5.0	0,98880	0,95973	3667.0	3.0	45.0	572.0
0.9546246851422694						

A.2 10-fold cross-validation

fold	accuracy	f1-score	TP	FN	FP	TN
Dataset_1	0,00000	0,00000	0.0	0.0	0.0	0.0
RandomForestClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,95348	0,91556	1432.0	0.0	95.0	515.0
2.0	0,95103	0,91379	1412.0	0.0	100.0	530.0
3.0	0,93340	0,87857	1414.0	0.0	136.0	492.0
4.0	0,96278	0,92910	1468.0	2.0	74.0	498.0
5.0	0,98580	0,97609	1421.0	2.0	27.0	592.0
6.0	0,98776	0,98014	1400.0	17.0	8.0	617.0
7.0	0,94757	0,90815	1405.0	0.0	107.0	529.0
8.0	0,97697	0,95959	1436.0	2.0	45.0	558.0
9.0	0,97158	0,95076	1423.0	7.0	51.0	560.0
10.0	0,95737	0,92294	1433.0	0.0	87.0	521.0
0.9334706311824196	0,00000	0,00000	0.0	0.0	0.0	0.0
DecisionTreeClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,66161	0,62869	766.0	666.0	25.0	585.0
2.0	0,72674	0,23562	1398.0	14.0	544.0	86.0
3.0	0,69050	0,64811	828.0	586.0	46.0	582.0
4.0	0,63957	0,59956	755.0	715.0	21.0	551.0
5.0	0,71303	0,66893	864.0	559.0	27.0	592.0
6.0	0,67238	0,64129	775.0	642.0	27.0	598.0
7.0	0,67124	0,63710	781.0	624.0	47.0	589.0
8.0	0,67663	0,63536	806.0	632.0	28.0	575.0
9.0	0,66340	0,63282	762.0	668.0	19.0	592.0
10.0	0,70701	0,20690	1365.0	68.0	530.0	78.0
0.5534369233291011	0,00000	0,00000	0.0	0.0	0.0	0.0
SVC	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,97698	0,96067	1421.0	11.0	36.0	574.0
2.0	0,96817	0,94694	1397.0	15.0	50.0	580.0
3.0	0,97502	0,95823	1406.0	8.0	43.0	585.0
4.0	0,97747	0,95863	1463.0	7.0	39.0	533.0
5.0	0,97551	0,95840	1416.0	7.0	43.0	576.0
6.0	0,97698	0,96151	1408.0	9.0	38.0	587.0
7.0	0,97991	0,96743	1391.0	14.0	27.0	609.0
8.0	0,97501	0,95667	1427.0	11.0	40.0	563.0
9.0	0,97403	0,95602	1412.0	18.0	35.0	576.0
10.0	0,98236	0,97000	1423.0	10.0	26.0	582.0
0.9594502475108833	0,00000	0,00000	0.0	0.0	0.0	0.0
GaussianNB	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,57542	0,58297	569.0	863.0	4.0	606.0
2.0	0,57786	0,59263	553.0	859.0	3.0	627.0
3.0	0,58521	0,59647	569.0	845.0	2.0	626.0
4.0	0,56562	0,56284	584.0	886.0	1.0	571.0
5.0	0,57640	0,58829	559.0	864.0	1.0	618.0
6.0	0,58766	0,59480	582.0	835.0	7.0	618.0
7.0	0,60510	0,60987	605.0	800.0	6.0	630.0
8.0	0,57374	0,57971	571.0	867.0	3.0	600.0
9.0	0,59040	0,59339	595.0	835.0	1.0	610.0
10.0	0,58746	0,58967	594.0	839.0	3.0	605.0
0.5890645643387316	0,00000	0,00000	0.0	0.0	0.0	0.0
KNeighborsClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,97062	0,94872	1427.0	5.0	55.0	555.0
2.0	0,95103	0,91453	1407.0	5.0	95.0	535.0

3.0	0,95593	0,92386	1406.0	8.0	82.0	546.0
4.0	0,98188	0,96693	1464.0	6.0	31.0	541.0
5.0	0,98139	0,96880	1414.0	9.0	29.0	590.0
6.0	0,97209	0,95285	1409.0	8.0	49.0	576.0
7.0	0,96570	0,94234	1399.0	6.0	64.0	572.0
8.0	0,97893	0,96334	1433.0	5.0	38.0	565.0
9.0	0,97893	0,96414	1420.0	10.0	33.0	578.0
10.0	0,95590	0,92077	1428.0	5.0	85.0	523.0
0.9466288067084102	0,00000	0,00000	0.0	0.0	0.0	0.0
GradientBoostingClassifie	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,74094	0,43179	1312.0	120.0	409.0	201.0
2.0	0,78844	0,48571	1406.0	6.0	426.0	204.0
3.0	0,75808	0,45714	1340.0	74.0	420.0	208.0
4.0	0,73506	0,67311	944.0	526.0	15.0	557.0
5.0	0,76641	0,72056	950.0	473.0	4.0	615.0
6.0	0,70813	0,67035	840.0	577.0	19.0	606.0
7.0	0,76629	0,46704	1355.0	50.0	427.0	209.0
8.0	0,69525	0,65521	828.0	610.0	12.0	591.0
9.0	0,80255	0,75015	1033.0	397.0	6.0	605.0
10.0	0,79569	0,50999	1407.0	26.0	391.0	217.0
0.582107205691319	0,00000	0,00000	0.0	0.0	0.0	0.0
MLPClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,98727	0,97876	1417.0	15.0	11.0	599.0
2.0	0,97698	0,96348	1375.0	37.0	10.0	620.0
3.0	0,98972	0,98332	1402.0	12.0	9.0	619.0
4.0	0,98629	0,97574	1451.0	19.0	9.0	563.0
5.0	0,99119	0,98558	1409.0	14.0	4.0	615.0
6.0	0,97258	0,95645	1371.0	46.0	10.0	615.0
7.0	0,99020	0,98447	1387.0	18.0	2.0	634.0
8.0	0,98383	0,97328	1407.0	31.0	2.0	601.0
9.0	0,98383	0,97354	1401.0	29.0	4.0	607.0
10.0	0,98236	0,97092	1404.0	29.0	7.0	601.0
0.9745535571121133	0,00000	0,00000	0.0	0.0	0.0	0.0
fold	accuracy	f1-score	TP	FN	FP	TN
Dataset_2	0,00000	0,00000	0.0	0.0	0.0	0.0
RandomForestClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,97224	0,97199	601.0	29.0	5.0	590.0
2.0	0,96082	0,96053	593.0	32.0	16.0	584.0
3.0	0,64327	0,49363	575.0	0.0	437.0	213.0
4.0	0,91102	0,90644	588.0	1.0	108.0	528.0
5.0	0,92082	0,91499	606.0	0.0	97.0	522.0
6.0	0,92408	0,91629	623.0	0.0	93.0	509.0
7.0	0,96816	0,96837	589.0	29.0	10.0	597.0
8.0	0,91837	0,90859	628.0	0.0	100.0	497.0
9.0	0,96571	0,96749	558.0	35.0	7.0	625.0
10.0	0,97388	0,97472	576.0	23.0	9.0	617.0
0.8983040810215683	0,00000	0,00000	0.0	0.0	0.0	0.0
DecisionTreeClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,73878	0,77305	360.0	270.0	50.0	545.0
2.0	0,78531	0,80733	411.0	214.0	49.0	551.0
3.0	0,52163	0,18611	572.0	3.0	583.0	67.0
4.0	0,58694	0,43146	527.0	62.0	444.0	192.0
5.0	0,63184	0,47982	566.0	40.0	411.0	208.0

6.0	0,58939	0,30236	613.0	10.0	493.0	109.0
7.0	0,70367	0,76566	269.0	349.0	14.0	593.0
8.0	0,76082	0,79439	366.0	262.0	31.0	566.0
9.0	0,72653	0,78233	288.0	305.0	30.0	602.0
10.0	0,66612	0,74389	222.0	377.0	32.0	594.0
0.6066382944878324	0,00000	0,00000	0.0	0.0	0.0	0.0
SVC	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,96571	0,96517	601.0	29.0	13.0	582.0
2.0	0,96082	0,96007	600.0	25.0	23.0	577.0
3.0	0,98204	0,98302	566.0	9.0	13.0	637.0
4.0	0,96245	0,96401	563.0	26.0	20.0	616.0
5.0	0,96816	0,96892	578.0	28.0	11.0	608.0
6.0	0,97796	0,97763	608.0	15.0	12.0	590.0
7.0	0,96327	0,96302	594.0	24.0	21.0	586.0
8.0	0,97224	0,97199	601.0	27.0	7.0	590.0
9.0	0,96653	0,96824	559.0	34.0	7.0	625.0
10.0	0,96816	0,96917	573.0	26.0	13.0	613.0
0.9691255300734698	0,00000	0,00000	0.0	0.0	0.0	0.0
GaussianNB	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,71673	0,77305	287.0	343.0	4.0	591.0
2.0	0,71673	0,77394	284.0	341.0	6.0	594.0
3.0	0,71755	0,78954	230.0	345.0	1.0	649.0
4.0	0,71265	0,78298	238.0	351.0	1.0	635.0
5.0	0,70041	0,77105	240.0	366.0	1.0	618.0
6.0	0,68408	0,75429	244.0	379.0	8.0	594.0
7.0	0,69878	0,76571	253.0	365.0	4.0	603.0
8.0	0,67347	0,74747	233.0	395.0	5.0	592.0
9.0	0,71265	0,78191	242.0	351.0	1.0	631.0
10.0	0,70041	0,77191	237.0	362.0	5.0	621.0
0.7711862675695047	0,00000	0,00000	0.0	0.0	0.0	0.0
KNeighborsClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,96980	0,96924	605.0	25.0	12.0	583.0
2.0	0,96898	0,96796	613.0	12.0	26.0	574.0
3.0	0,93224	0,93268	567.0	8.0	75.0	575.0
4.0	0,95673	0,95688	584.0	5.0	48.0	588.0
5.0	0,96571	0,96512	602.0	4.0	38.0	581.0
6.0	0,95020	0,94700	619.0	4.0	57.0	545.0
7.0	0,97143	0,97081	608.0	10.0	25.0	582.0
8.0	0,95184	0,94883	619.0	9.0	50.0	547.0
9.0	0,97224	0,97289	581.0	12.0	22.0	610.0
10.0	0,97388	0,97411	591.0	8.0	24.0	602.0
0.9605516932565692	0,00000	0,00000	0.0	0.0	0.0	0.0
GradientBoostingClassifie	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,74776	0,79303	324.0	306.0	3.0	592.0
2.0	0,67837	0,75220	233.0	392.0	2.0	598.0
3.0	0,52898	0,20414	574.0	1.0	576.0	74.0
4.0	0,63755	0,48131	575.0	14.0	430.0	206.0
5.0	0,65959	0,51228	589.0	17.0	400.0	219.0
6.0	0,67265	0,50555	619.0	4.0	397.0	205.0
7.0	0,67837	0,75406	227.0	391.0	3.0	604.0
8.0	0,86286	0,85159	575.0	53.0	115.0	482.0
9.0	0,72571	0,78894	261.0	332.0	4.0	628.0
10.0	0,67918	0,76080	207.0	392.0	1.0	625.0

0.6403906832817295	0,00000	0,00000	0.0	0.0	0.0	0.0
MLPClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,98531	0,98502	615.0	15.0	3.0	592.0
2.0	0,93224	0,92960	594.0	31.0	52.0	548.0
3.0	0,98612	0,98695	565.0	10.0	7.0	643.0
4.0	0,97306	0,97463	558.0	31.0	2.0	634.0
5.0	0,94776	0,94847	572.0	34.0	30.0	589.0
6.0	0,98857	0,98831	619.0	4.0	10.0	592.0
7.0	0,93224	0,93112	581.0	37.0	46.0	561.0
8.0	0,98612	0,98582	617.0	11.0	6.0	591.0
9.0	0,97224	0,97377	560.0	33.0	1.0	631.0
10.0	0,96816	0,96887	579.0	20.0	19.0	607.0
0.9672580445142402	0,00000	0,00000	0.0	0.0	0.0	0.0
fold	accuracy	f1-score	TP	FN	FP	TN
Dataset_3	0,00000	0,00000	0.0	0.0	0.0	0.0
RandomForestClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,78020	0,52582	605.0	0.0	202.0	112.0
2.0	0,95212	0,92143	617.0	0.0	44.0	258.0
3.0	0,78455	0,51232	617.0	0.0	198.0	104.0
4.0	0,96844	0,95076	610.0	10.0	19.0	280.0
5.0	0,78020	0,50490	614.0	0.0	202.0	103.0
6.0	0,94342	0,91503	587.0	9.0	43.0	280.0
7.0	0,96409	0,94931	577.0	25.0	8.0	309.0
8.0	0,96732	0,95342	581.0	17.0	13.0	307.0
9.0	0,95643	0,92727	623.0	0.0	40.0	255.0
10.0	0,97821	0,96416	629.0	5.0	15.0	269.0
0.8124419424117321	0,00000	0,00000	0.0	0.0	0.0	0.0
DecisionTreeClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,73014	0,44643	571.0	34.0	214.0	100.0
2.0	0,70403	0,45161	535.0	82.0	190.0	112.0
3.0	0,71491	0,24713	614.0	3.0	259.0	43.0
4.0	0,60283	0,61376	264.0	356.0	9.0	290.0
5.0	0,68553	0,15249	604.0	10.0	279.0	26.0
6.0	0,45484	0,56245	96.0	500.0	1.0	322.0
7.0	0,58215	0,61600	227.0	375.0	9.0	308.0
8.0	0,45861	0,56134	103.0	495.0	2.0	318.0
9.0	0,78758	0,56570	596.0	27.0	168.0	127.0
10.0	0,52723	0,55894	209.0	425.0	9.0	275.0
0.4775848828162711	0,00000	0,00000	0.0	0.0	0.0	0.0
SVC	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,97824	0,96815	595.0	10.0	10.0	304.0
2.0	0,97933	0,96817	611.0	6.0	13.0	289.0
3.0	0,97388	0,95904	614.0	3.0	21.0	281.0
4.0	0,97280	0,95712	615.0	5.0	20.0	279.0
5.0	0,97497	0,96186	606.0	8.0	15.0	290.0
6.0	0,96083	0,94322	584.0	12.0	24.0	299.0
7.0	0,97171	0,95793	597.0	5.0	21.0	296.0
8.0	0,97712	0,96629	596.0	2.0	19.0	301.0
9.0	0,97821	0,96552	618.0	5.0	15.0	280.0
10.0	0,97712	0,96270	626.0	8.0	13.0	271.0
0.961000283910544	0,00000	0,00000	0.0	0.0	0.0	0.0
GaussianNB	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,53428	0,59470	177.0	428.0	0.0	314.0

2.0	0,56148	0,59900	215.0	402.0	1.0	301.0
3.0	0,59195	0,61220	248.0	369.0	6.0	296.0
4.0	0,53972	0,58570	197.0	423.0	0.0	299.0
5.0	0,54625	0,59078	201.0	413.0	4.0	301.0
6.0	0,56692	0,61804	199.0	397.0	1.0	322.0
7.0	0,55604	0,60694	196.0	406.0	2.0	315.0
8.0	0,54684	0,60456	184.0	414.0	2.0	318.0
9.0	0,53050	0,57704	193.0	430.0	1.0	294.0
10.0	0,54031	0,57287	213.0	421.0	1.0	283.0
0.5961832381601531	0,00000	0,00000	0.0	0.0	0.0	0.0
KNeighborsClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,97280	0,95935	599.0	6.0	19.0	295.0
2.0	0,97497	0,96134	610.0	7.0	16.0	286.0
3.0	0,96844	0,94974	616.0	1.0	28.0	274.0
4.0	0,97715	0,96398	617.0	3.0	18.0	281.0
5.0	0,96844	0,95026	613.0	1.0	28.0	277.0
6.0	0,97171	0,95912	588.0	8.0	18.0	305.0
7.0	0,97062	0,95624	597.0	5.0	22.0	295.0
8.0	0,97168	0,95820	594.0	4.0	22.0	298.0
9.0	0,97603	0,96207	617.0	6.0	16.0	279.0
10.0	0,96950	0,94982	625.0	9.0	19.0	265.0
0.9570119350819237	0,00000	0,00000	0.0	0.0	0.0	0.0
GradientBoostingClassifie	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,75626	0,48624	589.0	16.0	208.0	106.0
2.0	0,77476	0,51064	604.0	13.0	194.0	108.0
3.0	0,70294	0,18507	615.0	2.0	271.0	31.0
4.0	0,62024	0,62912	274.0	346.0	3.0	296.0
5.0	0,70294	0,18991	614.0	0.0	273.0	32.0
6.0	0,57563	0,62282	207.0	389.0	1.0	322.0
7.0	0,58215	0,62055	221.0	381.0	3.0	314.0
8.0	0,57190	0,61955	205.0	393.0	0.0	320.0
9.0	0,80610	0,58796	613.0	10.0	168.0	127.0
10.0	0,60131	0,60645	270.0	364.0	2.0	282.0
0.5058327010833037	0,00000	0,00000	0.0	0.0	0.0	0.0
MLPClassifier	0,00000	0,00000	0.0	0.0	0.0	0.0
1.0	0,97606	0,96584	586.0	19.0	3.0	311.0
2.0	0,98041	0,97106	599.0	18.0	0.0	302.0
3.0	0,99021	0,98517	611.0	6.0	3.0	299.0
4.0	0,98694	0,98007	612.0	8.0	4.0	295.0
5.0	0,98368	0,97553	605.0	9.0	6.0	299.0
6.0	0,98041	0,97256	582.0	14.0	4.0	319.0
7.0	0,95321	0,93250	579.0	23.0	20.0	297.0
8.0	0,96623	0,95238	577.0	21.0	10.0	310.0
9.0	0,99020	0,98492	615.0	8.0	1.0	294.0
10.0	0,94118	0,90847	596.0	38.0	16.0	268.0
0.9628506411801212	0,00000	0,00000	0.0	0.0	0.0	0.0

Appendix B

Experiment 5: model evaluation results

B.1 All samples and features

All

Accuracy	**F1-score**	**True Negative**	**False Positive**	**False Negative**	**True Positive**
Dataset 1					
SVC					
0,99031	0,78956	74408	620	122	1392
KNeighborsClassifier					
0,98550	0,70958	74076	952	158	1356
MLPClassifier					
0,98846	0,76781	74199	829	54	1460
Dataset 2					
SVC					
0,97887	0,64313	73468	1560	57	1457
KNeighborsClassifier					
0,97139	0,55775	72971	2057	133	1381
MLPClassifier					
0,98187	0,67885	73687	1341	47	1467
Dataset 3					
SVC					
0,98784	0,74845	74226	802	129	1385
KNeighborsClassifier					
0,96746	0,52234	72689	2339	152	1362
MLPClassifier					
0,98234	0,68233	73738	1290	62	1452
Dataset 4					
SVC					
0,99573	0,87920	75025	3	324	1190
KNeighborsClassifier					
0,99505	0,85926	75006	22	357	1157
MLPClassifier					
0,99664	0,90954	74993	35	222	1292
Dataset 5					
SVC					
0,99462	0,86829	74772	256	156	1358
KNeighborsClassifier					
0,99493	0,86574	74903	125	263	1251
MLPClassifier					
0,99586	0,89908	74813	215	102	1412
Dataset 6					
SVC					
0,99616	0,90134	74905	123	171	1343
KNeighborsClassifier					
0,99526	0,87303	74931	97	266	1248
MLPClassifier					
0,99707	0,92402	74956	72	152	1362
Dataset 7					
SVC					
0,99211	0,81438	74613	415	189	1325

KNeighborsClassifier					
0,98994	0,76072	74548	480	290	1224
MLPClassifier					
0,99423	0,85977	74745	283	159	1355

B.2 All samples and top 100 features

100 ALL

Accuracy	**F1-score**	**True Negative**	**False Positive**	**False Negative**	**True Positive**
Dataset 1					
SVC					
0,98950	0,77159	74380	648	156	1358
KNeighborsClassifier					
0,98031	0,63748	73710	1318	189	1325
MLPClassifier					
0,98633	0,73506	74045	983	63	1451
Dataset 2					
SVC					
0,97792	0,62922	73418	1610	80	1434
KNeighborsClassifier					
0,96112	0,47495	72220	2808	168	1346
MLPClassifier					
0,98457	0,70512	73949	1079	102	1412
Dataset 3					
SVC					
0,98729	0,73823	74197	831	142	1372
KNeighborsClassifier					
0,96397	0,49708	72421	2607	151	1363
MLPClassifier					
0,98197	0,67712	73715	1313	67	1447
Dataset 4					
SVC					
0,99540	0,86875	75025	3	349	1165
KNeighborsClassifier					
0,99467	0,84821	74994	34	374	1140
MLPClassifier					
0,99637	0,90086	75001	27	251	1263
Dataset 5					
SVC					
0,99447	0,86117	74807	221	202	1312
KNeighborsClassifier					
0,99382	0,83851	74841	187	286	1228
MLPClassifier					
0,99541	0,88746	74807	221	130	1384
Dataset 6					
SVC					
0,99557	0,88567	74890	138	201	1313
KNeighborsClassifier					
0,99450	0,85468	74883	145	276	1238
MLPClassifier					
0,99658	0,91173	74927	101	161	1353
Dataset 7					
SVC					
0,99177	0,80325	74626	402	228	1286

KNeighborsClassifier					
0,98990	0,75745	74562	466	307	1207
MLPClassifier					
0,99318	0,83439	74705	323	199	1315