

Tormod Haus Lien

# Hunting malicious scripts using machine learning

Master's thesis in Information Security

Supervisor: Geir Olav Dyrkolbotn

Co-supervisor: Felix Leder

June 2021

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



Norwegian University of  
Science and Technology



Tormod Haus Lien

# Hunting malicious scripts using machine learning

Master's thesis in Information Security  
Supervisor: Geir Olav Dyrkolbotn  
Co-supervisor: Felix Leder  
June 2021

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



Kunnskap for en bedre verden





# Hunting malicious scripts using machine learning

Tormod Haus Lien

CC-BY 2021/06/01



# Acknowledgements

We would like to thank our supervisors Geir Olav Dyrkolbotn (NTNU) and Felix Leder (Norton LifeLock) for their superb guidance and contributions during this master thesis.



# Abstract

Computer systems has become more and more crucial for our society, which means that these systems has become more valuable than ever before and are therefore prone to interactions from individuals with malicious intent. There are often many ways to attack a system, but one type that has gained a lot of popularity over the last couple of years are malicious scripts, also known as fileless malware. This is where PowerShell becomes relevant as a type of fileless malware and the use of malicious PowerShell scripts has increased immensely the last years. Most Windows systems nowadays have PowerShell integrated as it is intended for system administrators to make their life easier by e.g., automating tasks. The fact that system administrators use this tool is a statement to how powerful it can be, also part of its name, and many systems allow scripts to be executed without many restrictions. Many of these malicious scripts are hard to detect because of the fact that they abuse a legitimate tool but also because they use actions also performed in similar ways by e.g, administrators.

In this thesis we propose a method that uses NLP technology in order to parse and tokenize PowerShell code. The tokens kept for further feature extraction are the commands and variables. From the commands and variables we extracted character frequencies, minimum, maximum and average length. The frequency of a set of commands with the potential for malicious actions were also extracted. Lastly we extracted the verbs used in the commands in order to explore the frequencies of valid and invalid verbs compared to Microsoft's naming convention. With these features we tested the classification performance of a k-nearest neighbour(KNN) and a decision tree. All of our performance evaluations used 5-fold stratified crossvalidation and retrieved the mean scores. The best performing model was our KNN using all the features, which achieved an AUC score of 0.976 and the time used for training and testing with crossvalidation was 0.53 seconds. This means that our proposed method shows potential for being used as a filter for more complex and time consuming classification methods.



# Sammendrag

Datasystemer har blitt mer og mer avgjørende for samfunnet vårt, noe som betyr at disse systemene har blitt mer verdifulle enn noen gang før, og er derfor utsatt for interaksjoner fra personer med ondsinnede hensikter. Det er ofte mange måter å angripe et system på, men en type som har økt i popularitet er filløse skadevarer. Det er her PowerShell blir relevant som en type fil-løs skadevare, og bruken av ondsinnede PowerShell-skript har økt enormt de siste årene. De fleste Windows systemer har i dag PowerShell integrert, da det er ment for systemadministratorer for å gjøre livet deres enklere ved for eksempel å automatisere oppgaver. Det faktum at systemadministratorer bruker dette verktøyet, er erklæring om hvor kraftig det kan være, også en del av navnet, og mange systemer tillater at skript kjøres uten mange restriksjoner. Mange av disse ondsinnede skriptene er vanskelig å oppdage på grunn av det faktum at de misbruker et legitimt verktøy, men også på grunn av at de bruker handlinger som også utføres på lignende måter av for eksempel administratorer

I denne oppgaven forslår vi en metode som bruker NLP-teknologi for å analysere og tokenisere PowerShell-koden. Token som tas med videre for ytterligere trekk analyse og trekk utvinning er kommandoene og variablene. Fra kommandoene og variablene hentet vi ut tegnfrekvenser, minimum, maksimum og gjennomsnittlig lengde. Frekvensen til et sett med kommandoer med potensial for ondsinnede handlinger blir også hentet ut. Til slutt hentet vi ut verbene som ble brukt i kommandoene for å utforske frekvensen til gyldige og ugyldige verb sammenlignet med Microsofts navnekonvensjon. Med disse trekkene testet vi klassifiseringsevnen til en k-nærmeste naboer (KNN) og et beslutningstre. Alle evalueringer av modellenes evner brukte fem ganger stratifisert kryssvalidering og hentet ut gjennomsnittresultatet. Den modellen som ga best resultat var KNN modellen som brukte alle trekkene, og oppnådde en AUC-verdi på 0.976 og tiden som ble brukt til trening og testing med kryssvalideringen var 0.53 sekunder. Dette betyr at den foreslåtte metoden vår viser potensial for å bli brukt som et filter for mer komplekse og tidkrevende klassifiseringsmetoder.





# Contents

<b>Acknowledgements</b> . . . . .	<b>iii</b>
<b>Abstract</b> . . . . .	<b>v</b>
<b>Sammendrag</b> . . . . .	<b>vii</b>
<b>Contents</b> . . . . .	<b>ix</b>
<b>Figures</b> . . . . .	<b>xi</b>
<b>Tables</b> . . . . .	<b>xiii</b>
<b>Code Listings</b> . . . . .	<b>xv</b>
<b>1 Introduction</b> . . . . .	<b>1</b>
1.1 Topic covered by the project . . . . .	1
1.2 Keywords . . . . .	2
1.3 Problem description . . . . .	2
1.4 Justification, motivation and benefits . . . . .	3
1.5 Research questions . . . . .	3
1.6 Planned contributions . . . . .	3
1.7 Thesis Outline . . . . .	4
<b>2 Background</b> . . . . .	<b>5</b>
2.1 Malware . . . . .	5
2.1.1 Binaries . . . . .	6
2.1.2 Scripts . . . . .	6
2.2 Code analysis techniques . . . . .	9
2.3 Natural language processing . . . . .	10
2.4 Parsing . . . . .	10
2.5 Machine Learning . . . . .	11
<b>3 Related Work</b> . . . . .	<b>13</b>
3.1 Available Tools . . . . .	13
3.2 Malware detection . . . . .	13
3.3 What have we learned? . . . . .	16
<b>4 Methodology</b> . . . . .	<b>17</b>
4.1 Overview . . . . .	17
4.2 Scripts . . . . .	18
4.3 Feature extraction . . . . .	18
4.4 Feature Selection . . . . .	20
4.5 Classification . . . . .	22
<b>5 Results</b> . . . . .	<b>23</b>

5.1	Virtual environment specifications . . . . .	23
5.2	Feature extraction & Selection . . . . .	24
5.3	Classification . . . . .	33
<b>6</b>	<b>Discussion . . . . .</b>	<b>51</b>
<b>7</b>	<b>Conclusion . . . . .</b>	<b>59</b>
	<b>Bibliography . . . . .</b>	<b>63</b>

# Figures

2.1	Clean PowerShell example . . . . .	8
2.2	Malicious PowerShell command example . . . . .	9
2.3	Malicious PowerShell variable example . . . . .	9
4.1	Machine learning process overview . . . . .	17
4.2	Feature extraction in virtual environment . . . . .	19
5.1	Computer CPU used for feature extraction . . . . .	23
5.2	Memory allocated to the VM . . . . .	24
5.3	CPU cores allocated to the VM . . . . .	24
5.4	Count of malicious commands in dataset . . . . .	25
5.5	Count of command character in every file . . . . .	26
5.6	Count of variable characters in every file . . . . .	26
5.7	SelectKBest variable features with chi2 algorithm . . . . .	27
5.8	SelectKBest variable features with F_classif algorithm . . . . .	27
5.9	Variable Extra-trees classifier with random_state=0 . . . . .	28
5.10	Variable Extra-trees classifier with random_state=22 . . . . .	28
5.11	Variable Extra-trees classifier with random_state=42 . . . . .	29
5.12	Variable Extra-trees classifier with random_state=100 . . . . .	29
5.13	SelectKBest command features with chi2 algorithm . . . . .	30
5.14	SelectKBest command features with F_classif algorithm . . . . .	31
5.15	Command Extra-trees classifier with random_state=0 . . . . .	31
5.16	Command Extra-trees classifier with random_state=22 . . . . .	32
5.17	Command Extra-trees classifier with random_state=42 . . . . .	32
5.18	Command Extra-trees classifier with random_state=100 . . . . .	33
5.19	Tuning KNN for "all command features" . . . . .	34
5.20	KNN confusion matrix for "all command features" . . . . .	35
5.21	"All command features" AUC score for knn model . . . . .	35
5.22	Decision tree confusion matrix for "all command features" . . . . .	36
5.23	"All command features" AUC score for decision tree model . . . . .	37
5.24	Tuning KNN for "Command char and length" . . . . .	37
5.25	KNN confusion matrix for "Command char and length" . . . . .	38
5.26	"Command char and length" AUC score for knn model . . . . .	38
5.27	Decision tree confusion matrix for "Command char and length" . . . . .	39

5.28 "Command char and length" AUC score for Decision tree model . . .	39
5.29 Tuning KNN for "All variable features" . . . . .	40
5.30 KNN confusion matrix for "All variable features" . . . . .	40
5.31 "All variable features" AUC score for knn model . . . . .	41
5.32 Decision tree confusion matrix for "All variable features" . . . . .	41
5.33 "All variable features" AUC score for Decision tree model . . . . .	42
5.34 Tuning KNN for "Verb check features" . . . . .	42
5.35 KNN confusion matrix for "Verb check features" . . . . .	43
5.36 "Verb check features" AUC score for knn model . . . . .	43
5.37 Decision tree confusion matrix for "Verb check features" . . . . .	44
5.38 "Verb check features" AUC score for Decision tree model . . . . .	44
5.39 Tuning KNN for "Known commands" . . . . .	45
5.40 KNN confusion matrix for "Known commands" . . . . .	45
5.41 "Known commands" AUC score for knn model . . . . .	46
5.42 Decision tree confusion matrix for "Known commands" . . . . .	46
5.43 "Known commands" AUC score for Decision tree model . . . . .	47
5.44 Tuning KNN for "All features" . . . . .	47
5.45 KNN confusion matrix for "All features" . . . . .	48
5.46 "All features" AUC score for knn model . . . . .	48
5.47 Decision tree confusion matrix for "All features" . . . . .	49
5.48 "All features" AUC score for Decision tree model . . . . .	49
6.1 Verb action indicators from clean file . . . . .	56
6.2 Verb action indicators from malicious file . . . . .	57

# Tables

2.1	Confusion matrix . . . . .	12
4.1	Table of intended features to extract . . . . .	20
4.2	Table of intended features sets for classification . . . . .	22
5.1	Table of used software and versions . . . . .	24
5.2	Table of variable features . . . . .	30
5.3	Table of command features . . . . .	33
5.4	Classification scores . . . . .	50
5.5	Training and testing time . . . . .	50



# Code Listings

5.1	Testing and training split . . . . .	26
5.2	extra-trees model . . . . .	27
5.3	Shuffled stratified 5-fold crossvalidation . . . . .	34
5.4	Gridsearch for tuning the KNN . . . . .	34
5.5	Gridsearch for tuning the Decision tree . . . . .	36





# Chapter 1

## Introduction

This chapter presents the topic of the thesis, relevant keywords and description of the problems we are faced with. Further on it will present justifications, motivations and benefits of the thesis before diving into the research questions and planned contributions for the thesis.

### 1.1 Topic covered by the project

Computer systems has become more and more crucial for our society, which means that these systems has become more valuable than ever before and are therefore prone to interactions from individuals with malicious intent. One of the methods these individuals can use to exploit these systems are by using malware. Malware is defined by Microsoft [1] as "malicious applications and code that can cause damage and disrupt normal use of devices."

There are several types of malware out there, but one type that has gained a lot of popularity over the last couple of years are malicious scripts, also known as fileless malware. When looking at the different types of malicious scripts we can see that the popularity of using malicious PowerShell scripts has risen by an incredible 1,902 % over the past year according to McAfee's "Covid-19 threats report" [2]. PowerShell is a tool developed by Microsoft to support for example system administrators in automating tasks and administrating systems. It was made open-source and cross-platform in 2016 and as the name suggests it is a very powerful tool where you might be able to do a lot of changes to a system. Most of the systems we use are in some way or another in interaction with humans, which means that they are suspect to human errors. With that in mind we can say that the wide deployment of PowerShell and the .NET framework makes it an ideal attack method, and maybe most importantly PowerShell is a trusted application which means executing scripts often will be allowed to execute with impunity. This makes it possible for attackers to use the scripting language on its own to perform malicious actions, but they can also use PowerShell to inject payloads into running applications.

Today's antivirus software has problems with detecting these malicious Power-

Shell scripts, which means there is a need for new ways of trying to detect them. There has been a lot of different research done into using machine learning to classify malware, but according to Tajiri [3] there has been almost no research done into using Natural Language Processing (NLP) technology to classify PowerShell scripts. This paper will aim to fill a small part of this void by focusing on classifying malicious PowerShell by using NLP technology.

A co-student will cover obfuscated scripts in his thesis, which is why that is not a focus in this thesis. This thesis will focus on non-obfuscated scripts.

## 1.2 Keywords

Malware, Machine Learning, Static Analysis, Feature Extraction, Features Selection, Natural Language Processing (NLP), Information Security, PowerShell.

## 1.3 Problem description

Today's antivirus software struggles with detecting malicious PowerShell scripts since it is a type of fileless malware that takes advantage of the immense possibilities enabled by PowerShell. The amount of malicious PowerShell has increased to an amount where it is impossible for the analysts to be able to analyse every one of them. This means that a lot of potentially vital information about the different scripts might go undetected. The knowledge obtained from collected malware has proven essential in knowing how they work and which countermeasures needs to be implemented.

This type of analysis is not fully automated, and might never be, but there is potential of making the life of the analysts easier where as the backlog of scripts keeps increasing. If scripts that don't get analysed could reveal information about the bigger picture, e.g trends by the attackers, it also makes the other security jobs harder to do than it potentially could be.

A solution to the problem could be to hire enough analysts to analyse every malware out there, but that would be very costly as well as very difficult to find that many people with the required knowledge. Solving the problem without some form of automation is in other words almost impossible.

Since this paper will try to implement a solution using machine learning to help solve a tiny piece of this large problem there is also the general problems that occurs in any machine learning problem. The data needs to be of good quality, feature extraction needs to be performed in a sound manner and in the end a well supported classification needs to be done. Feature selection needs to be well reasoned and performed in a sound manner so that the features used to learn and classify are the ones of highest importance to the classification problem.

## 1.4 Justification, motivation and benefits

When defending our nations we have relied on information about the so called enemy. Most of us has probably heard the saying of Sun Tzu "Know thy enemy and know yourself...", and in this case we can improve our knowledge of the enemy or in other words malware creators. With a reliable way of classifying malicious PowerShell scripts it might be easier to notice trends and gather information that could gain other jobs within the field of information security.

In regards to the analysts it might be easier for them to see the bigger picture and connect dots since previously not analysed scripts can for example be more easily searched through. Time spent on analysing whether a script is malicious or not could also be spent on other tasks. If the classification is reliable and fast enough it might also be used as part of a live detection system.

By improving these aspects of the information security field we might be able to reduce the risk of attacks crippling companies. An example of a recent attack that utilized PowerShell in their attack was the ransomware that hit Norsk Hydro in 2019 [4]. This attack was an eyeopener for many people in Norway, and really shows how damaging and expensive these types of attacks can be.

The primary benefit of this thesis is more knowledge about new ways of classifying malicious PowerShell using NLP, while the secondary benefit would be methods that could be implemented to achieve better protection of systems that today is exploitable by the use of PowerShell.

## 1.5 Research questions

In an attempt to find a solution to a small part of this larger problem we need to focus on a set of research questions. These following questions has been made to clarify what this thesis actually wants to answer:

- How can NLP technology be used to perform static analysis on Malicious PowerShell scripts?
- What might be the best features for classifying malicious PowerShell when focusing on commands and variables?
- How well does the implemented solution perform when classifying Malicious PowerShell?
- What are the weaknesses of the implemented solution?

## 1.6 Planned contributions

As previously mentioned there is little research provided regarding the usage of NLP technology to classify malicious PowerShell. This thesis aims to help fill a small part of this void by exploring how NLP can be used to classify malicious PowerShell scripts. An implementation of the suggested solution will be performed and the performance of the solution will be reviewed.

## 1.7 Thesis Outline

This section gives an outline of the thesis, along with a description of what each chapter contains.

- Chapter 2: Background. This chapter aims to give the reader the required knowledge to follow the thought process and reasoning in later chapters
- Chapter 3: Related work. This chapter aims to explain what the state of the art research is when classifying malicious PowerShell. Additional relevant malware research will also be explored as PowerShell is a relatively new field.
- Chapter 4: Methodology. Here we aim to describe the methodology used when trying to answer the research questions as well as trying to explain our reasoning for the chosen method.
- Chapter 5: Results. In this chapter we aim to present the results achieved when executing the methodology explained in chapter 4.
- Chapter 6: Discussion. This chapter aims to discuss the results we presented in chapter 5 and try to draw lines and compare it to related research.
- Chapter 7: Conclusion. The final chapter aims to draw conclusions based on the discussions performed in chapter 6.

## Chapter 2

# Background

This chapter aims to give the required information about the topics **malware**, **code analysis techniques**, **natural language processing**, **parsing** and **machine learning** needed to understand this thesis.

### 2.1 Malware

Malware, also known as malicious code, is as previously stated "malicious applications and code that can cause damage and disrupt normal use of devices"[1]. There are many different types of malware out there, but a few examples are ransomware, key-loggers, Trojan horses and spyware.

A ransomware encrypts the files on a system so that the user can't get a hold of the data. You could say that the data is being held captive and to get it back the user needs to pay a ransom. This is what happened to Norsk Hydro in 2019[4]. One of the difficult aspects with this type of malware is that in recent history we have seen ransomware be used to attack systems, but there has been no intention of decrypting the data if a ransom is paid. This resulting in the data pretty much being deleted, and no way of getting it back.

Key-loggers records the keys used by a user when using i/o equipment, e.g. keyboard and mouse. The idea behind this type of malware is to collect the users interactions with a system in order to obtain important information, such as login credentials and banking information. Currently there is research being conducted into how typing behaviour can be used to protect systems. In theory this type of malware can be used to avoid such a security measure.

Trojan horse, or Trojan, is a type of malware that hides within a legitimate piece of software. This means that it is disguised as a normal program, but when installed it contains code that gives the attacker access to the system giving the possibility to steal data, monitor activity, install more malware and so on.

Spyware is malicious code obtaining information about the user and the system. This means that it collect information about browsing activity, logins, banking information and so on. Key-loggers can be defined as a type of spyware, but there are so much more information available than only key-strokes. Information about

the system can lead to the attacker finding vulnerabilities that can be exploited in other ways, for example how to get a Trojan on the system.

If we were to look at these methods from the attackers point of view there seems to be three different motivations behind the malware attacks:

- Financial motives
- Information gathering
- Destruction / Terrorism

Knowing the motives behind a attack can be useful when trying to figure out what parts of the system has been infected, when doing an investigation and maybe most importantly identifying the employees that are valuable targets for an attacker.

This thesis is focusing on malicious PowerShell, which means that it is import to know the difference between malicious binary and malicious scripts.

### **2.1.1 Binaries**

Binary files are files that are non-text files, or in other words contains data that is encoded in any other way than ASCII, UTF-8, UTF-16 and so on. Since it is encoded in a way that is not clear text we need a program to open it. People often think of binary files as only executable files, because when developing applications you most likely write your code in a higher level language before it is compiled into binaries. Looking at this from an attackers point of view, they would often like to install some piece of their own software on your system in order to run their malware. An example of a binary file that is not an executable, but still often important for malware analysts, are the ".dll" files. These files are libraries containing code and data for performing activities on a windows system, but are not executable programs on their own.

### **2.1.2 Scripts**

Scripts are similar to the application developing mentioned under binaries, but the difference is that the code written in a script is most likely not compiled. There are a few exceptions where a script is compiled and made into an executable, but this is not a focus in this thesis. Scripts are in other words clear text run by a script interpreter on the system. An example of scripts being used in your daily life could be when you are shopping online and add items to your cart. This has so far often been JavaScript interpreted by your web browser in order to give you a seamless shopping experience. We can say that the script automates actions to make your life easier. There are many different scripting languages, such as Python, Perl, PowerShell, Tcl, Microsoft's Visual Basic (VBA), command lines (Windows, Mac(Unix) or Linux) etc. The different script languages often have their own purpose and might not be designed to run scripts of an other language, but python can for example run Perl by using a built in function. This means that scripts can be run on any system that has an interpreter present that is capable of running

that specific script. If we again look at this from an attackers perspective, there are command lines present at both Microsoft, Unix and Linux systems. Many systems might have interpreters such as Python to perform machine learning and automate task like calculations, file manipulation etc. Nowadays all windows systems also comes with PowerShell pre-installed, which means that the system needs to be old or have PowerShell removed for it not to be there.

## PowerShell

Before PowerShell there was no single language for administrators to bridge different Microsoft and non-Microsoft tasks together. They had to be creative and use combinations of command prompt, scripting languages like Visual Basic Scripting Edition (VBScript) and software like Windows Script Host (WSH). In comparison the Unix and Linux administrators had C-shell and bash to rely on. This resulted in Microsoft developing PowerShell to cover the need for integration and automation, and the command line tool was released late 2006[5].

PowerShell became a integrated part of most windows systems when PowerShell 2.0 got released in 2009 and it was present on systems like windows 7 and windows server 2008 R2. At this stage features like PowerShell remoting, network file transfer, background jobs, script debugging, steppable pipeline and Windows PowerShell ISE got added. Moving on to 2012 version 3.0 got released and even more features got added, like scheduled jobs, and was integrated on for example windows 8. We are now on version 5.0 which is integrated on for example windows 10 systems and the tool has gotten more and more powerful for each released version[6].

System administrators can now use the tool for automating tasks such as user editing, network diagnostics, file manipulation, remotely interacting with other windows systems and so on. This makes it a very useful tools for windows system administrators, but it also means that adversaries can potentially do the same actions. And how would a system's defence mechanisms know the difference between the actions of a system administrator and an adversary when they both use PowerShell and not other arbitrary code?

To understand this thesis it is also important to know what PowerShell code looks like, and that is why we now will take a look at small parts of benign and malicious powershell script from the dataset. In Figure 2.1 we can see that the benign script starts with the line "Import-module servermanager". "Import-module" is the command and "servermanager" is the parameter used in the command. This is an example of how Microsoft intended PowerShell commands to be verb-noun pair. Import is the verb and module is the noun, which then indicates what type of action is being performed. Microsoft uses a list of approved and recommended verbs when creating commands, also known as cmdlets, and .NET classes. "Import" is one of these approved verbs and looking at several benign community created scripts they also try to mostly use this naming convention for commands. We have also been part of a project where we developed a PowerShell tool for

network log analysis, and this naming convention was also used in that project. It is not something script developers have to use in order for the scripts to work, but from our experience it is considered best practice to use it in order to have easily understandable code. The remaining parts of the script is a lot of if statements depending on the variable that was just created. The variable is called "version" as we can see by the \$ sign and it being assigned the value of operating system version.

```

import-module servermanager
$version=[System.Environment]::OSVersion.Version
if ($version.Major -lt 6 )
{
$webserver=Get-WindowsFeature -Name FS-NFS-Services
}
ElseIf(($version.Major -eq 6) -and ($version.Minor -le 1))
{
$webserver=Get-WindowsFeature -Name FS-NFS-Services
}
else
{
$webserver=Get-WindowsFeature -Name FS-NFS-Service
}
if(!$webserver.Installed)
{
import-module servermanager
if ($version.Major -lt 6 )
{$ret=Add-WindowsFeature FS-NFS-Services}
ElseIf(($version.Major -eq 6) -and ($version.Minor -le 1))
{$ret=Add-WindowsFeature FS-NFS-Services}
else
{$ret=Add-WindowsFeature FS-NFS-Service}
return $ret.Success
}
else
{return $webserver.Installed}

```

**Figure 2.1:** Clean PowerShell example

Figure 2.2 shows how a malicious scripts creates a function called "de". When a function like this is called later on in the script it works the same way as the command mentioned previously, but instead you use "de (params)". The difference between function and command might be confusing, but when our parser reads the script it classifies the "de" as a command when executed. This is only one example of how malicious scripts might not follow the same naming convention for commands, but there were also examples in the dataset where malicious scripts used the approved verbs followed by a noun.

Now that we understand commands a bit better we can explore the variables a bit as well. Previously we saw in Figure 2.1 how variables often are represented in benign scripts, and we can see that the variables makes sense and describes what it is. From personal experience and what we have seen so far from other scripts, it is also normal to use short variables like shown in Figure 2.2 where the variable is only one letter. We can now compare this to what the variables are in Figure 2.3



```
function de([String] $b, [String] $c)
{
$a = "RfcTS4gecPfsVW800GmGJQStTpZdJVdw5UdQl
zB/fr0xdnlyCK4wt7+SShso2+9Cq5s9c5boV+ydt2y:
dpfiy/VjBG7BVZ25ymwPUOpt2USV3ccBADv3Y1+Ebai
HSoPWkm6c7nyT1EpKZzGI51TIItcLI3FCYqVoSePA7zl
/K7536/NTm9nim8wF7iP51w59IF76i9+S0RT+0CnvX'
```

Figure 2.2: Malicious PowerShell command example

which is a small part of another malicious script. Almost all the lines in Figure 2.3 creates or edits a variable of some sort, and we can see that the names does seem to be a bit more random. Just looking at the first variable we can see that it is called "wjqd" and it's value is set to be something which is base64 encoded. Once again it is important to note that not all malicious scripts use variables like this, but one of the evasion mechanisms often used to avoid signature detection is code randomly changing the names of the variables and commands.

```
$wjqd = [System.Convert]::FromBase64String("Po/wCsslUaKzG1A0cTxTmC9TYtR/wuPrwCoy49tEz29/JJH1Dx4mYt6fYmDsYK/SKyEOj4jWoX45hhwWzI
mfEL0h4TIOfn/aJvB5Cra+VidSylvkoG2yFhcuAU7o6zg3Uum3GAVQ2tQ/R9j51YXeBB/HseAPTcsm94d0t1M9mfAtITLL9MR1wmmQBEdtcQGHgz1Kwmvh6giib
edUdQ5uRcgKb3yIAw9ggKOUHMFUsHXv4vxfmKannI/mI1J00X41B03feYbvRIUWKFOttkQwbDjz/TBfoF9vjXFsi0qkz2N+MRHNq0HyuiNV/x6cavHmtyfeBeUS7l
PFPSiMjRtrTf0QWGHndTQ0f6E15QiqdwnmsBt00S6YnXN1QZ5RrrnM3St3qp5/kjUSSix6Q1ZbmbEXG5uJEbqay0TvCwFraEfvd5XbqCDK601y/hZPbIn2070ri
$aypxd = New-Object "System.Security.Cryptography.AesManaged"
$gmzqhsnpq = [System.Convert]::FromBase64String("Ro6Zi2XwkFag6heBicnTx13EqQWnBwt97IzVgF3mPqs=")
$aypxd.IV = $wjqd[0..15]
$aypxd.Padding = [System.Security.Cryptography.PaddingMode]::PKCS7
$aypxd.Mode = [System.Security.Cryptography.CipherMode]::CBC
$aypxd.BlockSize = 128
$aypxd.KeySize = 256
$aypxd.Key = $gmzqhsnpq
$cogjuzgc = New-Object System.IO.MemoryStream(,$aypxd.CreateDecryptor().TransformFinalBlock($wjqd,16,$wjqd.Length-16))
$wrks = New-Object System.IO.MemoryStream
$bxngcotcc = New-Object System.IO.Compression.GzipStream $cogjuzgc, ([IO.Compression.CompressionMode]::Decompress)
$bxngcotcc.CopyTo($wrks)
$bxngcotcc.Close()
$aypxd.Dispose()
$cogjuzgc.Close()
$ujjwiqhm = [System.Text.Encoding]::UTF8.GetString($wrks.ToArray())
IEX($ujjwiqhm)
```

Figure 2.3: Malicious PowerShell variable example

## 2.2 Code analysis techniques

**Static analysis** is a form of analysis where you collect information from binaries or source code by decompressing or unpacking instead of running the malware [7]. When dealing with scripts you most often will not have to decompress or unpack the code since it is most likely not compiled. This means you can for example look at features like opcodes[8], which previous students at NTNU has done, string signatures[9], byte sequence[10] and control flow graphs[11]. An example of how it is possible to do this is using a program such as PeStudio to extract features, e.g. strings, and use these features and a machine learning method to detect malware. These static approaches struggle when the malware is obfuscated, but they do have the advantage of not running the malware which might take longer time and require more resources and not to mention potentially infecting a system.

**Dynamic analysis** is a form of analysis where you collect activities from API calls

or system calls when running the malware. This approach has the benefit of being able to handle obfuscation in a better way, but on the other hand it requires the execution of the malware which means potentially infecting a system. If the infected system is a isolated virtual environment this won't be a problem, but you still need to use the resources to run it and since many malware sleeps at the start to avoid detection you might need to let it run for a good while as well. There exists a good amount of research covering this approach on malware in general, e.g Ki et al. [12] who used the sequence of API calls and DNA sequence alignment algorithms to detect malware based on the sequence pattern.

## 2.3 Natural language processing

Natural language processing (NLP) is a field within linguistics and computer science where the goal is to process and analyze large amounts of natural language data. A computer is not designed to understand the type of language we use when we are writing or talking, and this is what nlp technology tries to solve. The NLP technology has improved drastically over the last decades, highly due to the improvements of machine learning(ML). In the early beginning of NLP they used rules in order to perform their tasks, so when the ML improved these rules also improved. The last years we have seen new NLP technology taking advantage of the newest and most complex ML in order to better obtain semantic and syntactic information from the analysed text. These new and complex NLP models have resulted in text-to-speech applications, chat-bots, topic segmentation, machine translation, text generation and so on. Since our problem is classifying malicious and benign PowerShell by analysing commands and variables it falls under nlp technology because we will be locating commands and variables in a large amount of text. From the commands we will also try to locate the verbs and nouns in order to further analyse the commands. When trying to understand language we can divide it into syntactic and semantic understanding. Syntax is the understanding of structure, e.g., finding verbs and nouns, whereas semantics is the understanding of meaning. Semantics is seen as the hardest to properly implement, but has seen improvements over the last years with the use of word-embeddings to create dictionaries and deep learning to try to understand the meaning of words, sentences and documents.

## 2.4 Parsing

Parsing is the process of analyzing a string of symbols and understanding understanding what it means. When using this technology within computer science and code analysis it means understanding the syntax and creating tags or tokens that represent the different parts of the code. As an example we can say that a script file is a string of symbols, but we want to know what parts of the string are variables, commands, comments, parameters etc. This is where the parsing technology

makes it possible to obtain information from the string and create new useful information. Depending on your problem description and what type of information you need you could extract the desired parts of the larger original string in order to gain information focusing only on the most relevant parts for your problem. This is in other words a type of natural language processing.

## 2.5 Machine Learning

To understand the implemented solution and the argumentation it is important to understand what machine learning really is. In some cases it might seem very complex, but Nils J. Nilson once said that "A learning machine, broadly defined, is any device whose actions are influenced by past experiences." [13]. This definition is basically what this thesis is trying to achieve. We want a machine that based upon previous knowledge about malicious and benign PowerShell, can classify unknown PowerShell scripts.

In order for a machine to learn anything we need features that it can use to learn. The first step is in other words to perform a feature extraction. Examples of this can be length of longest word, amount of signs, size of the file and so on. Next step is to decide which of these features are relevant by performing a feature selection. A known saying amongst machine learning experts is "garbage in, garbage out", and if we feed the machine learning algorithm lots of features that are not relevant it will only act as noise. When the extraction and selection is completed we are ready to start feeding the data to an algorithm.

The different machine learning methods can be divided into two main categories, which is supervised and unsupervised learning. Supervised learning is when you have a dataset that is labeled, which means that the learning algorithm has the answer key to use when training and evaluating the performance of the model. In our case this will be files labeled as malicious or benign. We can divide supervised learning even further based on the type of supervising variable, but based on our problem classification methods are of our interest. The most popular classification methods are decision trees and rules, nearest neighbor classifiers, support vector machines, Bayesian classifiers, discriminant functions and neural networks. Unsupervised learning on the other hand does not use labeled data. These algorithms are used to find patterns and connections on its own. If we were to use this in our problem it would try to classify the PowerShell files totally on its own by trying to find connections in the provided features. The unsupervised methods can be further divided into clustering and association rules [13]. There are also methods that is in between the two main categories, but those are not relevant for this thesis.

Overfitting is a dangerous aspect of machine learning, and it is important to know why we want to prevent it. When training a model on a dataset it can be tempting to use all the data you have in your possession in order to achieve the best possible result. The problem you then are faced with is the fact that your model might have learned the noise and inaccuracies in the data, and it might negatively impact

the performance when handling unseen data. Many machine learning algorithms require special techniques to prevent overfitting, but in binary classification problems the stratified cross validation is frequently used[13].

When evaluating how a method works it is useful to know how a confusion matrix works as well as an Area under ROC (Receiver Operating Characteristic) curve. A confusion matrix when dealing with a two-class problem like ours could look as simple as table 2.1:

	P	N
P	TP	FN
N	FP	TN

**Table 2.1:** Confusion matrix

First we have positive(P) which in our case would mean malicious, and we have negative(N) which means benign. True positives(TP) shows how many of the malicious files were classified as malicious. False positives(FP) shows how many of the benign files were classified as malicious. False negatives(FN) shows how many of the malicious files were classified as benign. True negatives(TN) shows how many of the benign files were classified as benign. From this matrix we can then utilise the following measures[13]:

- $Sensitivity = \frac{TP}{TP + FN}$
- $Accuracy = \frac{TP + TN}{TN + FP + FN + TN}$
- $Recall = \frac{TP}{TP + FN}$
- $Precision = \frac{TP}{TP + FP}$
- $F1 = \frac{2 * Recall * Precision}{Recall + Precision}$

Based on your classification problem it is possible to chose the most important measures and highlight these as long as a sound reasoning is present. The other measure previously mentioned was the Area under ROC curve, which shows the relation between the true positive rate (sensitivity) and false positive rate (1-specificity). ROC curves can also be used for a misclassification cost analysis[13].

## Chapter 3

# Related Work

This chapter aims to describe what currently is considered as state of the art within the research field of classifying malicious PowerShell scripts. When doing a search for "malicious powershell" in Oria, which is the electronic library used by NTNU, i get 28 hits. 11 of them are research papers, and the oldest one of them is from 2018. There was nothing older than 2018, except a book that in 2013 had one sentence warning the reader about how NuGet packages can contain PowerShell code which then runs under the same privileges as the visual studio application[14]. That sounds similar to how we in recent years have seen malicious PowerShell being added to PDF files and sent by email in order to attack companies when users open the PDF. After also searching google scholar it is clear that the research area of classifying malicious PowerShell is in a early phase.

### 3.1 Available Tools

Since this are of research is relatively uncharted there are not many openly available tools specifically designed for PowerShell analysis. The search for such tools resulted in the discovery of two different tools, "PowerShellRunBox"[15] and "PSOneTools"[16]. The difference between the two is that PSOneTools is a parser and will not execute the script, while PowerShellRunBox is a sandbox debugging tool and falls under dynamic analysis.

### 3.2 Malware detection

In 2018, Hendler et al. [17], proposed a method of using character embedding and convolutional neural networks (CNNs) to classify PowerShell commands. After processing their set of scripts they ended up with 66,388 distinct PowerShell commands, where 6,290 were labeled as malicious and 60,098 were labeled as clean. This is a very imbalanced dataset, and in order to get it more balanced they duplicated the malicious commands 8 times. Achieving a 1:1 ratio between malicious and benign commands they reduced the risk of over-fitting which is a known risk

when training a neural network on a small number of samples. They also state that the length of commands might be a good indicator to whether a command is malicious or not. Very long commands is one of the weaknesses when using a neural network for this task, since every character of a command will be fed into this network. This means that the classifier can't evaluate the entirety of commands that are longer than the network is wide, in this case commands longer than 1,024. To deal with this problem they truncate the commands that are too long before sending the allowed amount through the network. This paper also focuses a lot on obfuscation and states that the casing used in a command can be a good indicator for whether or not a command is obfuscated as well as malicious or benign. The result they achieved for detecting malicious commands achieved AUC scores in the range 0.985 - 0.990.

In 2019, Hendler et al. [18], proposed a method that seems to be taking the previous paper a step further by also obtaining information about the semantics in PowerShell. By using Word2Vec they were able to use euclidean distance to cluster commands and aliases reducing the dimensionality of the tokens extracted from the scripts. They do not state exactly which tokens they extract, but from the looks of it they are focusing on commands and the parameters used in those commands. Rare tokens are removed by setting a frequency threshold at 100 occurrences. Once again we have the limitation of only sending through a set amount of tokens and characters. The difference from the previous paper is that they now have tokens that are words, which means that only a set amount of words can be sent through the neural network. Once again they use a CNN before max pooling the output to reduce the dimensionality of the output. The character embedding is performed similarly to the previous paper sending the individual characters of tokens through the network. These two outputs are then concatenated before sent through a Bidirectional Long short-term memory (LSTM) which is a type of recurrent neural network (RNN). The limit is on 2000 tokens, and they chose to take the 2000 first tokens to send through the network. No more specific information about the tokens is provided, and a token can be e.g. commands, variables, comments and if-statements. They evaluated 10 different deep learning detection models where all achieved a AUC score above 0.987, and 0.995 at its best. The TPR for the best model was reported to be 0.922. When testing this model on a test set containing files acquired up to five months later they achieved a TPR of 0.894 at its best. The dataset used was relatively large, above 100,000 scripts, but the ratio between malicious and benign are very unbalanced. This resulted in not so many malicious scripts, and as in many other papers read for this thesis they use k-fold crossvalidation.

Mimura et al.[19] presented in 2021 a method for performing static detection of malicious PowerShell based on word embeddings. The features they extracted from the malicious PowerShell were word occurrences and Doc2Vec. Word occurrences was a selection of the most frequent words in both malicious and benign files, while Doc2Vec is a NLP tool that represents a document as a vector. It is a generalization of the Word2Vec tool, which looks at a set amount of words, but since it

focuses on the document and not each word specifically it is less memory heavy and doesn't have the same limitation in regards to max set of words or tokens. For classification they used SVM, RF, XGB and CNN. Their dataset consisted of 480 malicious and 5324 benign files. They did split their data into known and unknown data for training and testing, but as we can see the ratio between malicious and benign is highly imbalanced. When presenting their result they only present recall and f1 scores, and the best recall was at 0.990 while the best f1 was at 0.995. Another interesting aspect of this paper is that they present time used for training and testing their models, and achieved a required time of 0.9 seconds. Reading all these different papers on how deep learning could be used to achieve good results is both motivating and frustrating for someone at the entry level of machine learning. This resulted in a search for a way to soften the transition into deep learning, which led to a paper where Sunoh Choi[20] explains how k-nearest-neighbor (KNN) can be used as a fast screening to classify files since it is a much faster method. KNNs can in other words be faster, but also might be more unreliable as it is way less complex. Choi was analysing pe-files, but the research done here is still applicable when analysing PowerShell. As a result Choi states that the deep learning method increased its detection rate by 25% using the KNN method before the deep learning. The test data was 6000 files and the KNN reduced the detection time by 67% because it is faster and less complex than deep learning.

One of the supervisors for this thesis recommended looking into decision trees in addition to the KNN, and Patil et al. [21] describes how they extracted 4 different types of features from URL strings and used it to classify about 52 thousand URL strings as either malicious or benign. Achieving an AUC score of 0.998 at its best. The features extracted was both numerical, e.g. lengths of different aspects of the string, and binary like checks for "known" malicious strings. These 4 types of features had again a lot of sub features adding up to slightly above 100 different features. Even though this was performed towards URL strings the same methods could be applied to commands and variables as strings part of a script.

Fass et al.[22] proposed in 2019 a static pre-filter for malicious JavaScript detection. They used five different ways of abstracting code, which was Tokenizer, Parser, abstract syntax tree(AST) from parser, control flow graph(CFG) from AST and program dependence graph from CFG, in order to extract features. When classifying the JavaScripts they used two different layers in order to use the first layer to classify as many files as possible, while the second layer tries to classify the ones the first layer couldn't. They used a total dataset of 270,000 samples, and the first layer classified 93% of the dataset with an accuracy of 99,73%. The second layer classified another 6.5% with an accuracy over 99%. This left under 1% of the samples to be sent to additional analysis. They tested a set of different classifiers, (SVM, Bernoulli naive bayes, multinomial naive Bayes, and random forest), in order to select the one achieving the best result. The one with best result for their classification problem was the random forest. The best result they achieved was accuracy of 99.44%, FPR of 0.33% and 0.8% false-negatives.

### 3.3 What have we learned?

These papers are only a few of those we read for this thesis, but these became most relevant in the end. The most obvious notion about the PowerShell research is the focus on using deep learning for classification. These type of classifiers have the upside of being able to handle complex tasks, but the downside is that they are more computationally heavy as well as being harder to understand. Both the KNN and the decision tree are easier to implement as well as to understand and explain. We learned that one of the initial ideas we had about performing a verb check against Microsoft's approved verbs does not seem to have been performed by anyone. One very important thing we learned was the importance of looking at the result from different sides. A company might be more worried about legitimate scripts being classified as benign since this would interrupt the business, while a researcher like us might prefer to classify all the malicious scripts correctly and use a "better safe than sorry" mindset. Since the use of KNN and decision tree has been successfully used within other types of malware classification it might also be applicable to PowerShell and can help speed up the computation time used by deep learning methods as well as increasing their classifying performance. When comparing achieved results with related research it seems to be good practice to use the AUC score. The paper by Fass et al.[22] contained several steps that we also had in mind, but as a master thesis it might be too much to implement all the steps they presented. A good approach could be to approach the classification problem in a similar way and see how much we are able to do with the time we have. We also learned that obfuscation is highly relevant and several of the papers seemed to indicate that obfuscation also could be synonymous with maliciousness. It is worth mentioning that this thesis is done as part of a larger project and it might be a good idea for the reader to also read our co-students paper which focuses on obfuscation.



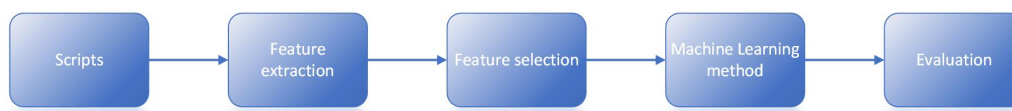
## Chapter 4

# Methodology

This chapter aims to describe the methodology used when trying to answer the research questions previously stated. It will also try to explain our reasoning for choosing this specific method. This way it is possible for others to repeat the research on the same dataset and achieve the same results, or on a new dataset and achieve comparable results. The main research question was how nlp-technology can be used to classify benign and malicious PowerShell scripts. From the related work chapter we can see how the state of the art methods looks like it takes advantage of deep learning also known as neural networks. The neural networks used in the most recent researched seemed to be using LSTMs, but as part of a previous course at NTNU we researched what the newest within deep learning was. We found the newest technology to be encoder-decoder models, also known as transformers, such as BERT[23] which was released by google in 2018. Implementing a deep learning solution is very tempting, but with the limited time of a master thesis and the fact that deep learning is very hard to understand and explain we decided to focus on less complex models. During the same research we saw even highly experienced professionals mention BERT as a black box. Because of this we have decided to implement a KNN and a decision tree in order to evaluate how our features perform as basis for classification.

### 4.1 Overview

Firstly it is important to gain an overview of the entire machine learning process we plan to perform before diving into the specifics. In Figure 4.1 we can see our planned process for this thesis.



**Figure 4.1:** Machine learning process overview

The first step is to acquire a set of scripts, both malicious and benign as well as hopefully a good ratio between the two. Next step is the feature extraction where we need to figure out a way to extract the features we have had in mind and only barely tested at this stage. Further on we need to perform a feature selection as there might be features that only would create noise for the signal of more important features. When we then hopefully know which features to use we need to implement the desired machine learning method as previously mentioned a KNN and a decision tree. The final step will be to evaluate the results achieved, compare it to the results of related research and finally point out weaknesses and potential improvements.

## 4.2 Scripts

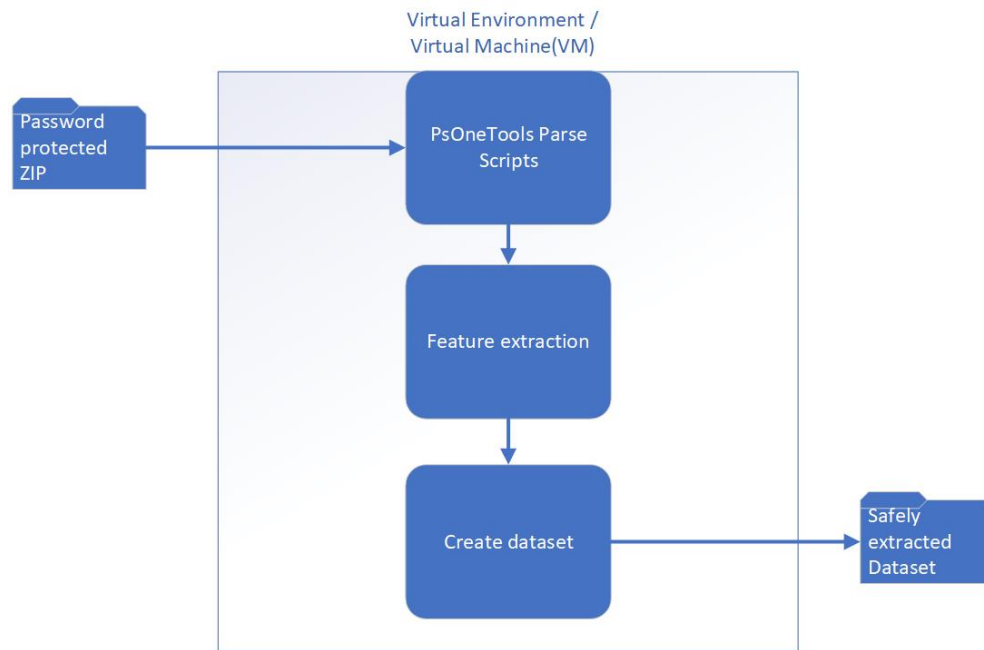
We have acquired a total of 1,725 PowerShell script files, where 808 of them are benign and 917 of them are malicious. This means that we have achieved one of our goals of having a relatively balanced ratio between malicious and benign. Compared to the related research this amount of files is on the lower end, and it will be important to use cross-validation in order to get a proper estimate of the learning quality[13]. Several of the research papers we read[21][18] used VirusTotal[24] in order to make sure that their files were malicious and benign. In the course "Computational forensics" we learned that splitting the test data in 80% for training and 20% for testing is good practice which is why we will perform this split as part of this thesis.

The files we have acquired are a courtesy of Norton LifeLock, who are considered as one of the leading companies within the field of cybersecurity, as part of a project agreement and NTNU cooperation. With this in mind we feel that it is a reasonable assumption that a company of this stature have correctly labeled the files. To our knowledge these are scripts from the wild and gives us a good representation of the malicious and benign scripts that are roaming out there. We do not know if these files are collected over a long period of time, but based on the date stamps we assume that these are not collected over a long period of time. The reason this is important is because the model will learn what the malicious files looks like right now, and new changes might not be detected at the same rate.

## 4.3 Feature extraction

Our proposed method uses static analysis to extract features from both the benign and malicious PowerShell files. Since we are handling malicious files it is important to use a safe environment where there is no risk for spread, and if infected can be reverted to a safe version. This is why we will use a virtual environment created with VirtualBox[25]. In Figure 4.2 we can see how we safely are going to extract the features in the virtual environment. We can also see that the parser used is PSOneTools[16] which is an open source community created module to Power-

Shell. This is an advanced parser turning PowerShell code into detailed tokens. The intention of the parser is to analyze or error check your own code, but it is also possible to use it to collect features from malicious files. Since we are performing static analysis we will not execute the malicious code, but accidents can happen and a virtual environment as shown in Figure 4.2 is still best practice.



**Figure 4.2:** Feature extraction in virtual environment

One of the struggles with detecting PowerShell is how creative the writers can be when writing their code. This means that we as defenders also need to be creative when extracting features. In order to be creative we need to take a look at malicious and benign files to get ideas relevant to PowerShell and the extraction possibilities that are available. Previously we saw that Hendler et al.[17] used the characters in PowerShell commands in order to classify them into benign or malicious. They removed the characters that did not frequently occur in the commands, but in this thesis we will gather all the characters used before evaluating them using during the feature selection phase. Under the background chapter we explored the commands and variables, and we saw indicators that might suggest that variables can be used in the same fashion. From our previous PowerShell experience in addition to the research done for this thesis it seems like the verb-noun naming convention could be used as features. That is why we intend to extract features that are numerical values of how many commands either use valid verbs or not. Since we don't know all the characters present in the commands and variables it is impossible to create a table at this point which shows all the features. Another idea we had is to create a word cloud for the most frequently used commands in the malicious files. We have a suspicion that invoke commands might

Feature	Description
Command characters	Count all characters present in the commands
Variable characters	Count all characters present in the variables
ValidVerbs	count the commands that use valid verbs in the script
InvalidVerbs	count the commands that use invalid verbs in the script
TotalVerbs	count of commands that used a verb-noun naming
InvalidTotal	InvalidVerbs divided by TotalVerbs
ValidTotal	ValidVerbs divided by TotalVerbs
Malicious commands	Create a word cloud and find the most used commands
MaxCom	Max length of commands used in the script
MinCom	Min length of commands used in the script
AvgCom	Average length of commands used in the script
MaxVar	Max length of variables used in the script
MinVar	Min length of variables used in the script
AvgVar	Average length of variables used in the script
ErrorCount	Count of syntax errors detected by parser

**Table 4.1:** Table of intended features to extract

be a valuable feature, because that is one way of executing PowerShell code in a remote fashion, but we don't know before learning more about the data. All the features planned to be extracted from each file is presented in Table 4.1.

## 4.4 Feature Selection

One very important aspect of this thesis will be feature selection. The amount of characters in the ascii table is 128, and from exploring some of the acquired scripts there is a good chance we will face most of the characters from that table. In addition to this we will also face non-ascii characters. We are in other words faced with many features creating a high dimensionality. Kononenko et al.[13] states that one of the problems with high-dimensional datasets is that, in many cases, not all the measured variables are important when trying to understand the underlying phenomena of interest. Dyrkolbotn et al.[26] describes three groups of feature selection methods:

- Ranking
- Subset
- Construction

As a quick explanation we can describe feature ranking as a way of scoring how important the features are individually. This means that any relations between the features are not accounted for. Since it computes one and one feature it is a computationally efficient method. Subset selection can be used to cover the area that feature ranking can't, which is how well a set of features work together in-

stead of individual performance. Construction methods are linear and non-linear dimension reduction techniques.

Our goal is to identify those features that individually can be used to classify malicious and benign PowerShell. The reason is that features found to individually be of importance can then be added to existing methods. Feature ranking is in other words what we are looking for and as it also is the most computationally efficient method we maintain our idea of implementing a light weight classification method.

All of the feature extraction will be performed by using PowerShell and PSOn-eTools. The feature selection will be performed using python and a tool called scikit-learn[27]. In the course "Computational forensics" we learned that it is best practice to implement several feature rankings in order to get a good picture of which features actually score well. After reading the documentation of scikit-learn we have decided to implement two feature selection methods and one ensemble method, where all three are feature ranking methods. The first selection method is using the SelectKBest method with the `f_classif` algorithm and the second using the same method, but with the `chi2` algorithm. With the `f_classif` we will compute the ANOVA f-value for the provided feature and in the end get a ranking. The `chi2` algorithm is used when we have booleans or frequencies data, which we have, and we will from this algorithm get a ranking based on what feature are most likely to be irrelevant for the classification[28]. The last method is the extra trees classifier which is a set of decision trees constructed from the training sample. Each node in the tree is provided with a random sample of k-features and splits the data based on the information-gain[29]. When performing these feature rankings it is important to ensure our selection is not based on a lucky split of the dataset. To avoid this problem we will use crossvalidation by performing four different splits of testing and training data. For the two SelectKBest methods we will get the top 20 features and select only the features which are present in top 20 for all four splits. In regards to the extra trees classifier we will use a set threshold for the four splits.

When we have the three different rankings we can start comparing them in order to get the best possible features for when we are implementing a classification method. We have the two main features being commands and variables. With the three rankings we can locate the features that scored the best in each of them. The information we want to find out is how well the different sets of features perform when trying to classify malicious PowerShell. From the desired features we will create the following subsets and evaluate their performance in order to test how our own features perform compared to the ones inspired by related work:

Feature set	Description
All command features	command chars, lengths and verb check
Command char and length	Command char and lengths features
All variable features	variable chars and lengths
Verb check features	only the verb check features
Known commands	commands from word-cloud and ErrorCount
All features	all command and variable features

**Table 4.2:** Table of intended features sets for classification

## 4.5 Classification

We now have six different feature sets and need to evaluate how they perform. To do this we have chosen to implement a KNN and a decision tree. The reason for choosing these two is as previously mentioned the way they can be used as a lightweight classification filter to increase the accuracy and computation speed of a deep learning method. They are also easily understandable, and as part of a master thesis we found it best to explore white box methods that we would be able to explain. Since most of the PowerShell research we found was focusing on deep learning, we found this approach to be a good idea as well as a potential building block if we in the future were to work further on this project. In the end we will have twelve different results that we will have to evaluate. The evaluation will be performed by looking at the confusion matrix and Area under ROC curve(AUC) score. The two evaluation methods are described under the background chapter. Once again it is important to use crossvalidation in order to get a sound evaluation of the models. Stratified crossvalidation is often used when dealing with classification problems[13], which is why we will use a five-fold stratified crossvalidation in order to evaluate the results.

## Chapter 5

# Results

This chapter aims to present the results achieved when executing the previously explained methodology. By presenting the achieved results we can later on answer our research questions to the best of our ability and justify our conclusions by referring to the results. We are in other words going to draw conclusions and discuss the results in later chapters.

### 5.1 Virtual environment specifications

We are not performing any computational evaluations in the virtual environment, but we started the testing trying to gather all the data in one json file and got faced with memory problems. This forced us to use one json file per PowerShell file. If someone were to recreate or keep working on this project it could be useful to know the resources our virtual environment got assigned. The cpu on the computer hosting the virtual machine(VM) is shown in Figure 5.1, and is a 8th gen i7 with 4 cores(8 logical). This means that parts of this cpu is what the virtual machine will be using when performing the feature extraction.

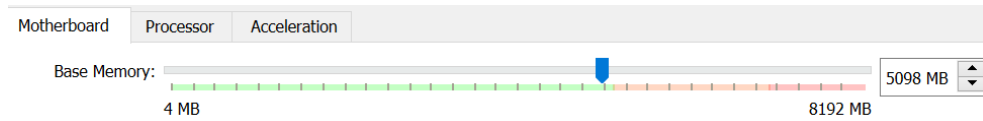
Systemtype	x64-based PC
System-SKU	LENOVO_MT_81C4_BU_idea_FM_YOGA C930-13IKB
Processor	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, 1992 Mhz, 4 kjerne(r), 8 logiske prosessor(er)
BIOS-versjon/-dato	LENOVO 8GCN30WW, 13.07.2018

**Figure 5.1:** Computer CPU used for feature extraction

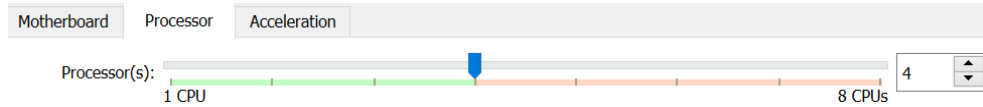
The allocated memory to the virtual machine is shown in Figure 5.2, and as we can see it is 5098 MB while the host computer has 8GB. The VM is also allocated 50GB virtual storage.

Lastly we can see in Figure 5.3 that the VM has 4 cores allocated out of the 8 logically processors present at the host computer. The VM was running a windows 10 OS since we intended to use PowerShell for feature extraction

When performing the feature selection and implementation of classification models we used another computer, which had a 4th gen i7 cpu with 4 cores(8 logical)



**Figure 5.2:** Memory allocated to the VM



**Figure 5.3:** CPU cores allocated to the VM

as well as 16 GB memory. The software used when doing this thesis, and its versions, is shown in Table 5.1.

Software	Version	Source
PowerShell	5.1.19041.610	[30]
PSOneTools	2.4	[16]
VirtualBox	6.1.2.35662	[25]
Python	3.8	[31]
Visual Studio Code	1.56.2	[32]
Pandas	1.0.3	[33]
scikit-learn	0.23.1	[34]

**Table 5.1:** Table of used software and versions

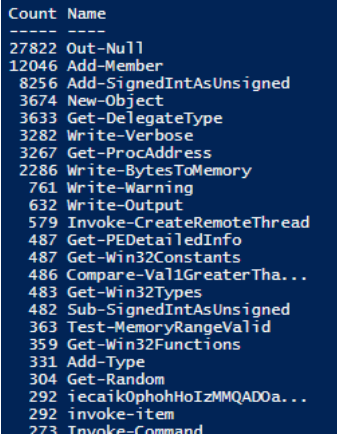
## 5.2 Feature extraction & Selection

This section describes the results achieved when performing feature extraction and selection. The reason for presenting them under the same topic is to make it easier to follow the process of the thesis. Extraction and selection is also closely related as we might not want to extract features we find to be less important when trying to create a efficient extraction process.

The first feature extraction performed in this thesis was the word-cloud approach towards the malicious commands. It is important to note that this is a count of how many times these commands were present in total. The result of the most used commands in the malicious files can be seen in Figure 5.4. From these commands we were looking for commands that to us might indicate malicious behaviour, and not a way to better classify only this dataset, which is why we opted to not look for Add-Member and New-Object. Out-Null is the most used command, and it is used to hide the output instead of it being displayed on the screen. This could be a good method for attackers when trying not to be detected as the user won't see any actions on the screen. The next command we found to be suspicious was



the `Add-SignedIntAsUnsigned`, which is used to add 64-bit memory addresses. `Get-delegateType` is also one we wanted to add, since delegates are PowerShell objects used when invoking methods. `Write-Verbose` is another way of hiding output from the user of a system. It is meant to be used as a debugging method, and the messages written will only show up if the user specifically tells the computer to show verbose messages in a script. `Get-ProcAddress` retrieves the address of an exported function or variable from a specific dynamic-link library (DLL), and Sikorski et al.[7] explains how malware creators often use DLLs to load malicious code. `Write-BytesToMemory` is used when writing shellcode to a remote process which calls a DLL, which is a combination of two actions we previously have mentioned as potential indicators. We could go on finding potentially malicious commands, but we figured a threshold of 2000 sightings was a good starting point. In addition to this we found that since `out-null` and `write-verbose` was used so much, we added a feature for the `Write-Host` command as it is the straight opposite and prints messages to the screen. This extraction was performed before the last extension of the dataset in an attempt to get an idea of malicious behaviour in general and not this dataset specifically.



Count	Name
27822	Out-Null
12046	Add-Member
8256	Add-SignedIntAsUnsigned
3674	New-Object
3633	Get-DelegateType
3282	Write-Verbose
3267	Get-ProcAddress
2286	Write-BytesToMemory
761	Write-Warning
632	Write-Output
579	Invoke-CreateRemoteThread
487	Get-PEDetailedInfo
487	Get-Win32Constants
486	Compare-Val1GreaterTha...
483	Get-Win32Types
482	Sub-SignedIntAsUnsigned
363	Test-MemoryRangeValid
359	Get-Win32Functions
331	Add-Type
304	Get-Random
292	iecaik0phohoIzMMQADOa...
292	invoke-item
273	Invoke-Command

**Figure 5.4:** Count of malicious commands in dataset

The next extraction we performed was the characters present in the variables and commands. As mentioned in the related work chapter Hendler et al.[18] were able to use these features for classification, although they used deep learning and the sequence of the characters would then have an impact. In Figure 5.5 we can see some of the characters present in the commands, and as we can see there are many different characters present. Both ASCII characters, Chinese characters and other signs seem to be present in the commands, and we can see at the bottom of the figure that it has seen 181 different characters. We can also see that the rows are 1,469 which is less than what we claimed to have in the dataset, and the reason is that this feature selection process was performed before the last expansion of the dataset.

In Figure 5.6 we can see the characters that were present in the variables, which



Feature scores for chi2:		Feature scores for chi2:		Feature scores for chi2:		Feature scores for chi2:	
Attribute	Score	Attribute	Score	Attribute	Score	Attribute	Score
59	/	1	E	1	/	1	/
1	E	3	R	6	T	3	E
3	R	4	D	3	R	6	T
6	T	6	T	4	D	4	D
4	D	17	N	14	I	17	N
17	N	14	I	17	N	17	N
14	I	10	A	10	A	14	I
0	S	0	S	0	S	0	S
10	A	11	L	0	S	10	A
11	L	7	O	11	L	11	L
7	O	15	C	7	O	15	C
15	C	8	P	8	P	7	O
8	P	52	%	15	C	8	P
38	[	38	[	52	%	52	%
44	space	44	space	38	[	38	[
40	-	40	-	40	-	40	-
2	U	48	]	44	space	44	space
48	]	2	U	2	U	48	]
18	F	18	F	18	F	2	U
35	2	35	2	20	Y	34	3

Figure 5.7: SelectKBest variable features with chi2 algorithm

F\_classif was the other algorithm to be used with the SelectKBest feature ranking method. The result using this algorithm is shown in Figure 5.8. Once again only those characters present in all the splits will be kept as features. Also here it seems to be mainly ASCII letters, but we can see that it also has the numbers 1, 2 and 3 present in all four splits:

Feature scores for f_classif:		Feature scores for f_classif:		Feature scores for f_classif:		Feature scores for f_classif:	
Attribute	Score	Attribute	Score	Attribute	Score	Attribute	Score
4	D	35	2	20	Y	35	2
35	2	34	3	4	D	34	3
21	W	4	D	35	2	4	D
34	3	21	W	14	I	20	Y
20	Y	20	Y	34	3	21	W
14	I	14	I	18	F	14	I
18	F	18	F	12	H	18	F
1	E	12	H	1	E	1	E
3	R	30	1	11	L	3	R
17	N	1	E	17	N	17	N
12	H	17	N	3	R	30	1
11	L	8	P	8	P	11	L
8	P	11	L	10	A	8	P
0	S	15	C	0	S	15	C
15	C	0	S	13	G	0	S
10	A	10	A	15	C	10	A
13	G	13	G	6	T	13	G
6	T	6	T	30	1	6	T
30	1	6	T	7	O	30	1
7	O	7	O			7	O

Figure 5.8: SelectKBest variable features with F\_classif algorithm

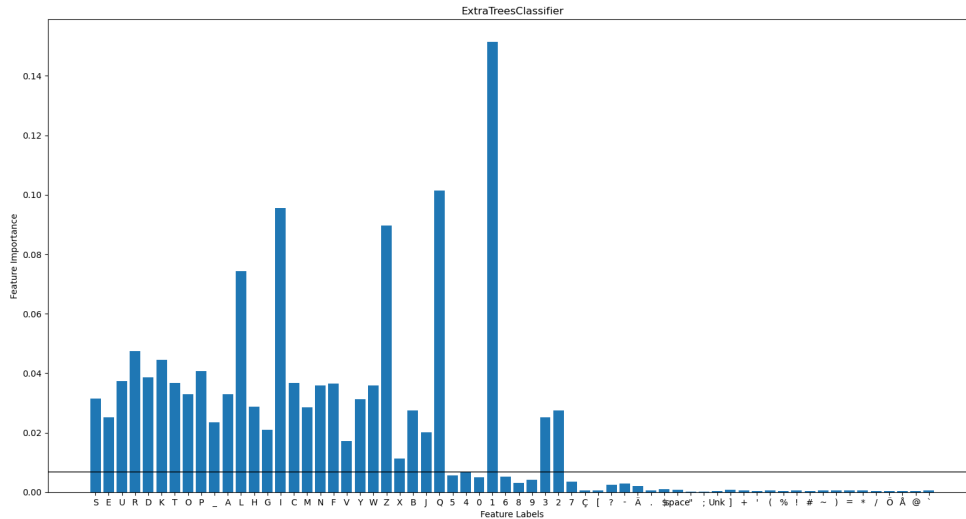
Finally we have the last feature ranking which is the extra-trees classifier ranking method which takes advantage of a set number of randomized decision trees in order to rank the features. Once again we split the data in the same four splits as previously. We set the threshold to be 0.004. Our model is shown in Code listing 5.2, and we can see that the number of trees being used is 100 and features selected at each split is 2. Entropy means that we use information gain in order to perform the splits. Random state is set in order to achieve repeatability:

## Code listing 5.2: extra-trees model

```
extModel = ExtraTreesClassifier(n_estimators=100, criterion='entropy',
max_features=2, random_state=42)
```

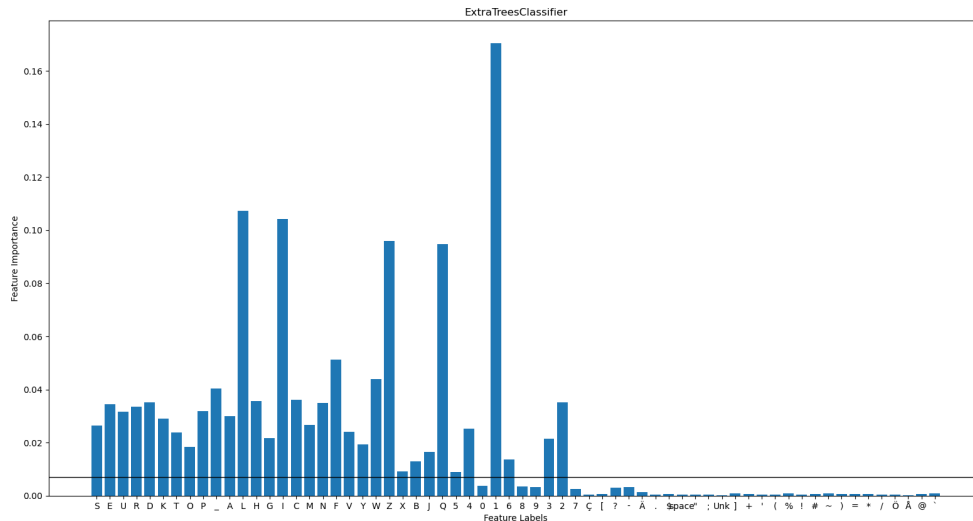
Results achieved from this feature ranking is shown in Figure 5.9, Figure 5.10, Figure 5.11 and Figure 5.12.

Figure 5.9 shows that all the ASCII letters are above the threshold as well as the underline sign and the numbers from 1 to 4:



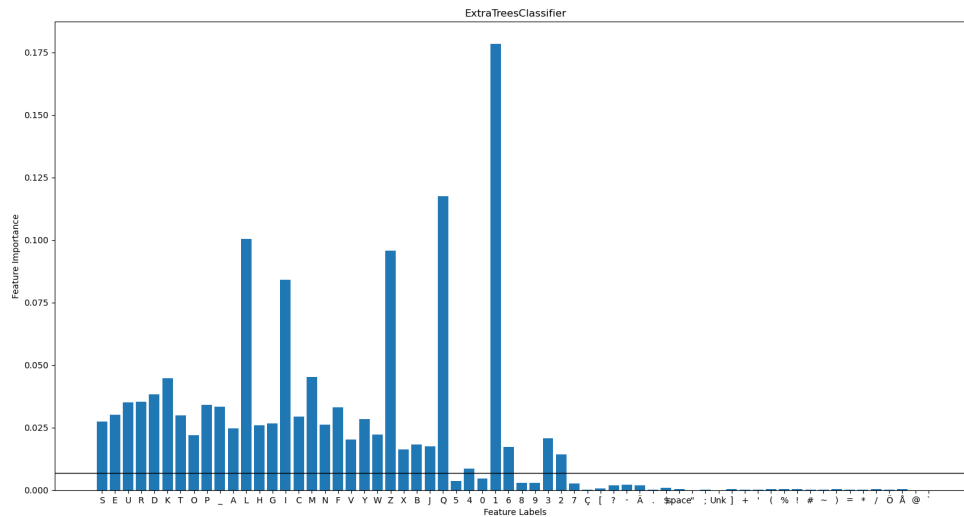
**Figure 5.9:** Variable Extra-trees classifier with `random_state=0`

Figure 5.10 gives us the same result as the previous split, but we can see that this time the number 5 and 6 has also reached above the threshold. Since these were not present at the previous split we do not note them down:



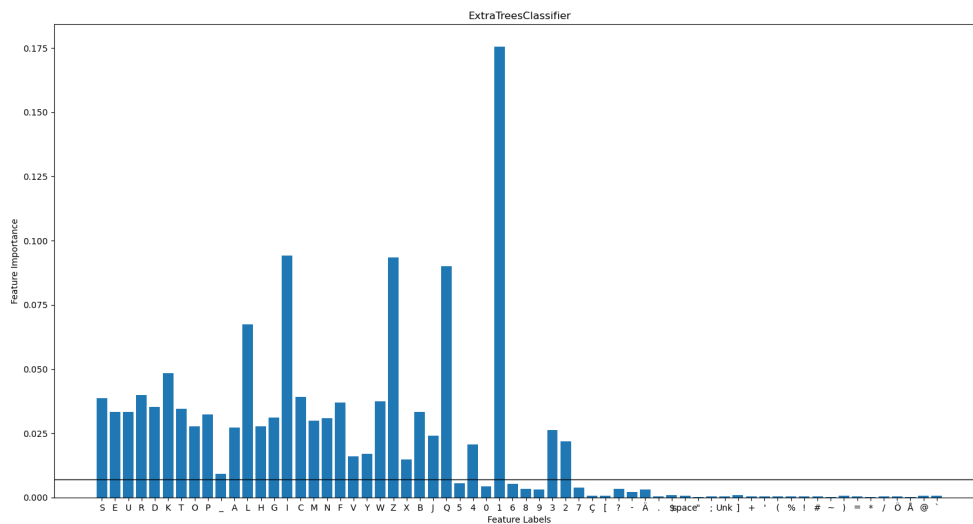
**Figure 5.10:** Variable Extra-trees classifier with `random_state=22`

Figure 5.11 is very similar to the first split as well, where the changes closest to the threshold is an increased importance to the number 4 and 6. But as stated earlier we will not take 6 with us further:



**Figure 5.11:** Variable Extra-trees classifier with `random_state=42`

In the last split, Figure 5.12, we can see that it has identically the same features over the threshold as in the first split:



**Figure 5.12:** Variable Extra-trees classifier with `random_state=100`

Same method as previously is once again used when we note the features that are over the threshold for all the four splits. To sum up the features we are going to use from the variables are presented in Table 5.2:

Feature	Description
varA - varZ (26 features)	count letters from a to z present in variables
var1 - var4 (4 features)	count number of 1, 2, 3 and 4 present in variables
varSkra	Count number of forward slashes in variables
varLine	count number of "-" in variables
varBrack	count number of "[" in variables
varUnder	count number of "_" in variables

**Table 5.2:** Table of variable features

Next step is to find the most important features from the command characters. Same method as used with the variable characters will now be used to find the most important command characters. When using the chi2 algorithm we got the result shown in Figure 5.13, and once again we note down the characters present in all four splits. Just like we saw with the variables we can see for the commands that it mainly is ASCII letters present in the top 20 as well as the line sign. The forward slash as well as the tilde sign being in three of the four splits, but we only want the ones present in all of them:

Feature scores for chi2:		Feature scores for chi2:		Feature scores for chi2:		Feature scores for chi2:					
Attribute	Score	Attribute	Score	Attribute	Score	Attribute	Score				
14	U	50114.757461	14	U	50114.757461	9	D	49218.453492	14	U	47102.241141
9	D	49033.772524	9	D	49033.772524	14	U	48742.976937	9	D	46704.041897
23	N	41719.061828	23	N	41719.061828	8	L	43128.048588	23	N	38630.334310
8	L	41086.028897	8	L	41086.028897	23	N	42313.392671	8	L	38076.260437
3	-	28466.386740	3	-	28466.386740	0	E	29461.404581	3	-	24954.144639
0	E	25278.286257	0	E	25278.286257	3	-	29198.384009	0	E	22028.798664
16	A	19711.011905	16	A	19711.011905	16	A	23636.922533	16	A	18061.275163
10	G	17233.963546	10	G	17233.963546	10	G	20180.132734	10	G	16080.175534
1	T	15714.034217	1	T	15714.034217	1	T	18071.415617	1	T	12530.882856
4	S	14734.637108	4	S	14734.637108	13	M	17541.196229	13	M	12421.939594
7	O	13077.288576	7	O	13077.288576	7	O	16174.217832	4	S	12366.536302
13	M	12921.248719	13	M	12921.248719	4	S	15959.648879	7	O	11296.953161
15	I	11408.757646	15	I	11408.757646	15	I	13686.844445	15	I	9667.094296
19	B	8466.270561	19	B	8466.270561	19	B	9983.885134	19	B	8066.324725
21	Y	5221.404474	21	Y	5221.404474	6	R	8921.534358	21	Y	4877.803233
6	R	5063.967970	6	R	5063.967970	21	Y	5401.508574	6	R	4270.035546
12	H	2875.950818	12	H	2875.950818	12	H	1942.218022	12	H	3201.888846
37	/	1694.614361	37	/	1694.614361	11	X	1453.356843	11	X	1736.750374
11	X	1630.333961	11	X	1630.333961	37	/	1383.635059	42	:	1597.887303
96	~	1579.913313	96	~	1579.913313	24	W	793.615627	96	~	1584.817337

**Figure 5.13:** SelectKBest command features with chi2 algorithm

Using the F\_classif algorithm we got the result shown in Figure 5.14, and we note down the features once again. There are some differences when we compare this result to the chi2 algorithm, but we can see that it still is mainly ASCII letters present. We can see that the question-mark sign as well as the line sign. There are also the numbers 1, 2 and 3 present in several splits, but only the number 3 is present in all of them:

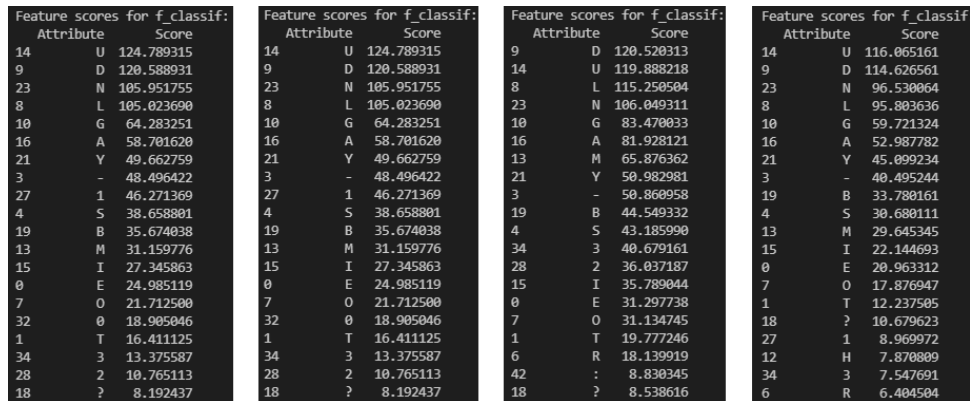


Figure 5.14: SelectKBest command features with F\_classif algorithm

The same extra-trees model we used for variables feature ranking is the same we now use for the command variables. Achieved results is shown in Figure 5.15, Figure 5.16, Figure 5.17 and Figure 5.18. In the first split we can see in Figure 5.15 that all the ASCII letters are above the threshold, but in addition to that we have the backslash, forwardslash and colon signs creeping above as well. Colon sign is the one close to the "Z". We also see the numbers 0, 1, 2 and 3 present:

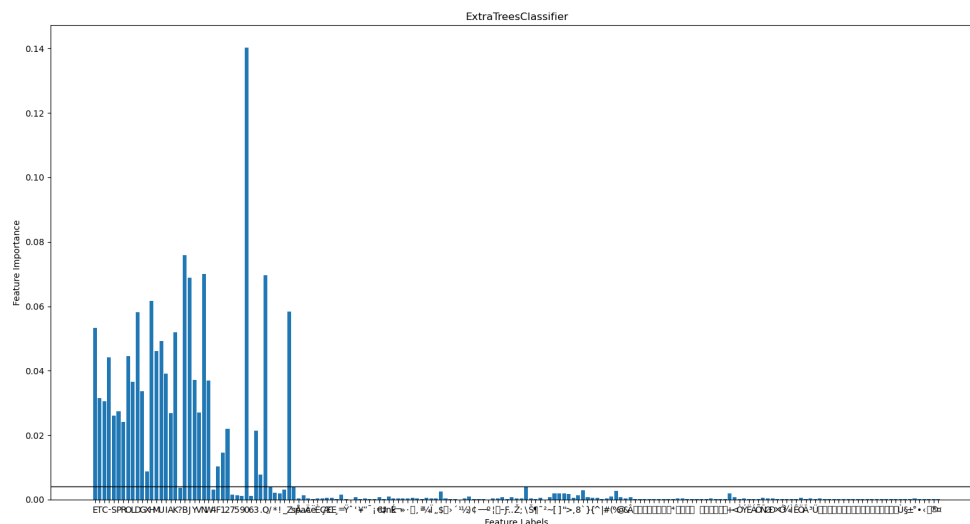


Figure 5.15: Command Extra-trees classifier with random\_state=0







evaluate the models performance. Code for executing this split was shown in Code listing 5.1. In order for the stratified k-folds crossvalidation to be repeatable we used random state equals to 42, and because our data is nicely ordered we use shuffle:

**Code listing 5.3:** Shuffled stratified 5-fold crossvalidation

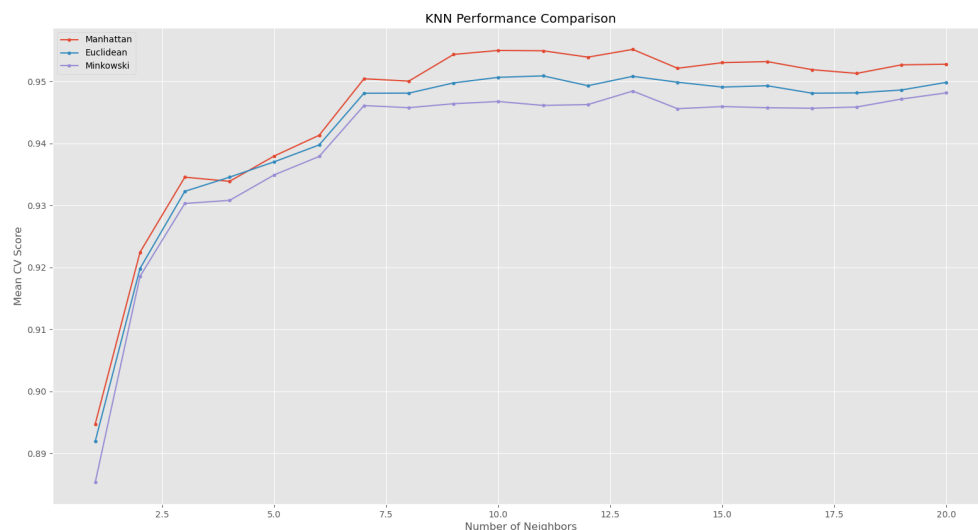
```
crossVal = StratifiedKFold(n_splits=5, random_state=42, shuffle=True)
```

The first feature set we are going to test is the one we called "All command features", which contained command character counts, command lengths and verb checks. To get the optimal model we need to tune the KNN, which we accomplish by using GridSearchCV from scikit and giving it information about which classifier we want to use and which parameters to test. It is important that the features sent to the KNN is scaled in order to avoid favouring large values. We have used the MinMaxScaler from the scikit library. The scoring we selected is the roc\_auc because we want to use this score in the end to compare our result with related work. Code used is shown in Code listing 5.4:

**Code listing 5.4:** Gridsearch for tuning the KNN

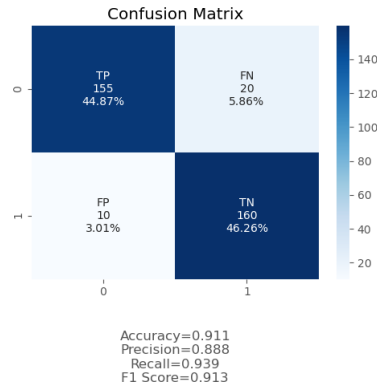
```
knnGridsearch = GridSearchCV(estimator=KNeighborsClassifier(weights='distance'),
param_grid=knn_params, cv=crossVal, verbose=1, scoring='roc_auc',
return_train_score=True, n_jobs=4)
```

In Figure 5.19 we can see how the grid-search gives us the optimal value for number of neighbours as well as which distance measure we should use. The best KNN with this feature set is as we can see 13 neighbors and Manhattan as distance measure. The auc score keeps rising until this point before it slowly seems to score lower the higher k its being presented with.



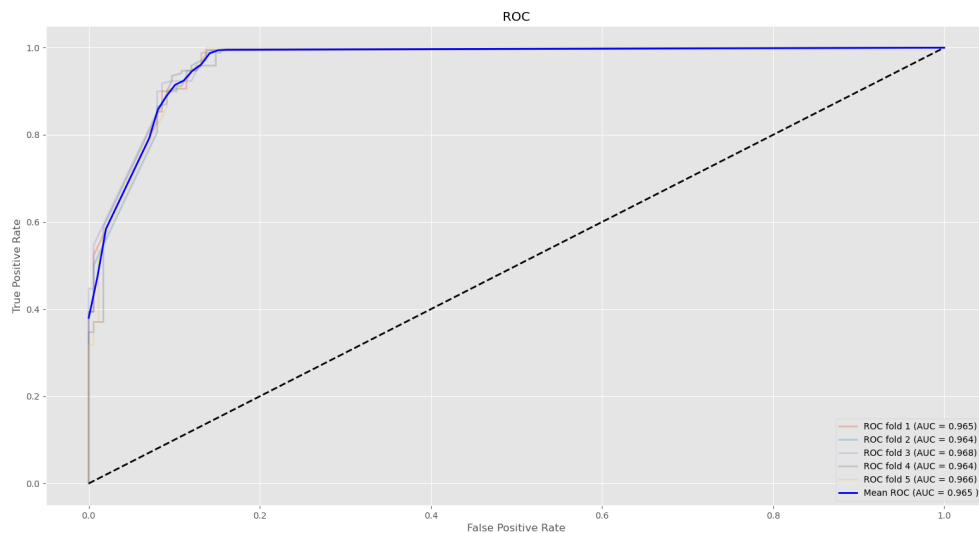
**Figure 5.19:** Tuning KNN for "all command features"

When implementing this model to the "All command features" set we get the confusion matrix shown in Figure 5.20. From the TP, FP, FN and TN values we can calculate accuracy, precision, recall and F1 score. The matrix is a mean of the matrices for each fold which means that the count displayed in Figure 5.20 is rounded to the closest integer. The count is not going to be used for further comparison, which is why we presented it as it would be in a real scenario as 0.x of a file does not make sense.



**Figure 5.20:** KNN confusion matrix for "all command features"

From each confusion matrix we can plot the ROC curve and calculate the auc\_roc score (AUC score) and then also get the mean result. The result we got from this model is shown in Figure 5.21, and we can see the AUC score for each of the stratified folds as well as the mean AUC score which we are most interested in. The mean AUC score achieved with this set and model was 0.965.



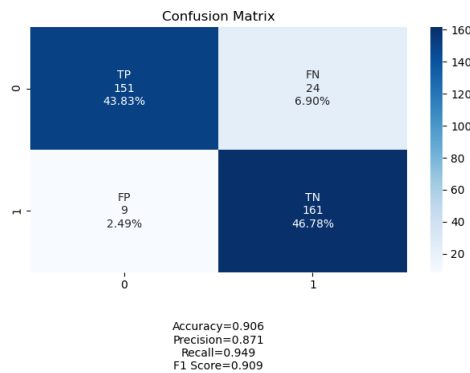
**Figure 5.21:** "All command features" AUC score for knn model

Next step is to do the same method, but for a decision tree as we want to compare the two models for this classification problem. To achieve repeatability with these decision trees as well we use the random state equals to 42. How we used the grid-search to tune the decision trees is shown in Code listing 5.5

**Code listing 5.5:** Gridsearch for tuning the Decision tree

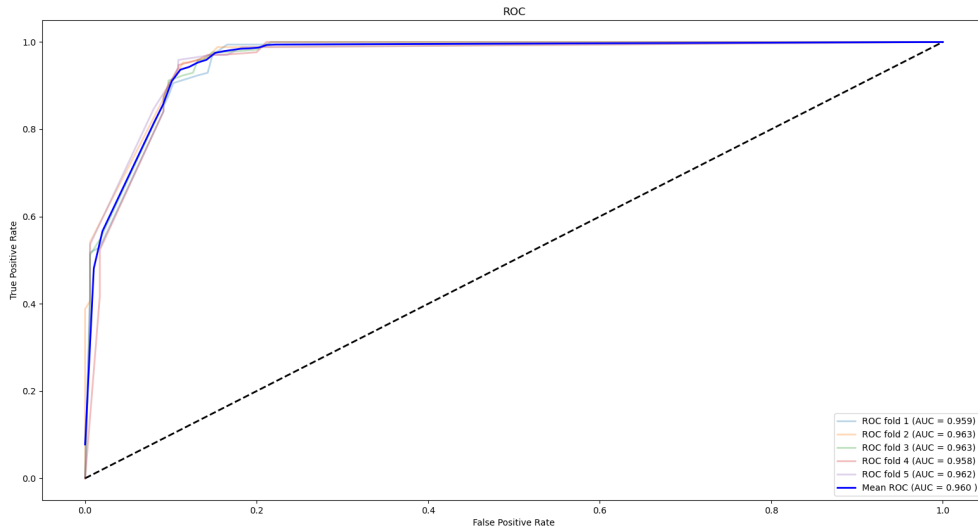
```
gsDT = GridSearchCV(estimator=DecisionTreeClassifier(random_state=42),
param_grid=paramsDT, cv=crossVal, verbose=1, scoring='roc_auc', n_jobs=4)
```

With the grid-search we tuned the decision tree in order to figure out the best parameters for this feature set. To calculate the quality of a split the best measure, also called criterion, was "gini", which is the gini impurity measure. The max\_depth, which sets the maximum depth of the tree, was best at 9. Min\_samples\_leaf, which sets the minimum number of samples required to be at a leaf node, was best at 11. The last parameter we tuned was the min\_samples\_split, which sets the minimum number of samples required to split an internal node, was best at 2. Mean confusion matrix achieved with the decision tree on the "all command features" set is shown in Figure 5.22:



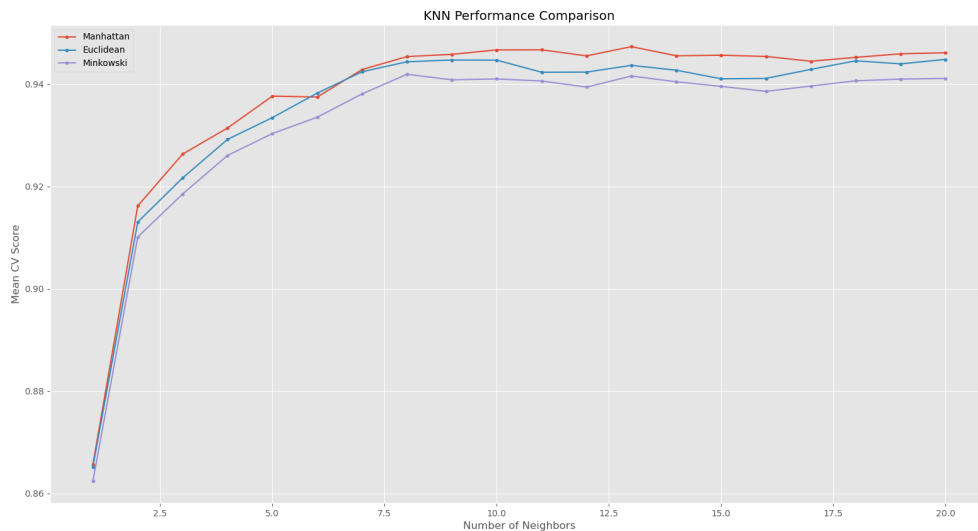
**Figure 5.22:** Decision tree confusion matrix for "all command features"

Figure 5.23 shows the AUC score of each stratified k-fold as well as the mean AUC. The mean AUC is 0.960, which is 0.05 lower than what we achieved with the KNN:



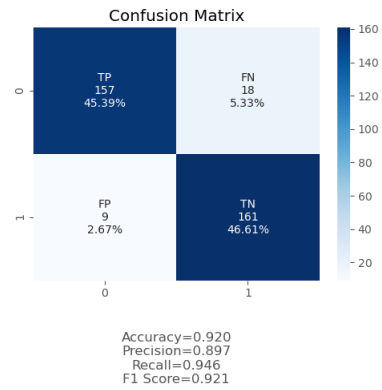
**Figure 5.23:** "All command features" AUC score for decision tree model

Moving on to the next feature set we have "Command char and length", which was inspired by related work using deep learning[18][17], focusing on command characters and min, max and avg length of commands. Tuning of the KNN model is shown in Figure 5.24, and we can see that it reaches a top at 13 neighbors with Manhattan measure before it flattens out:



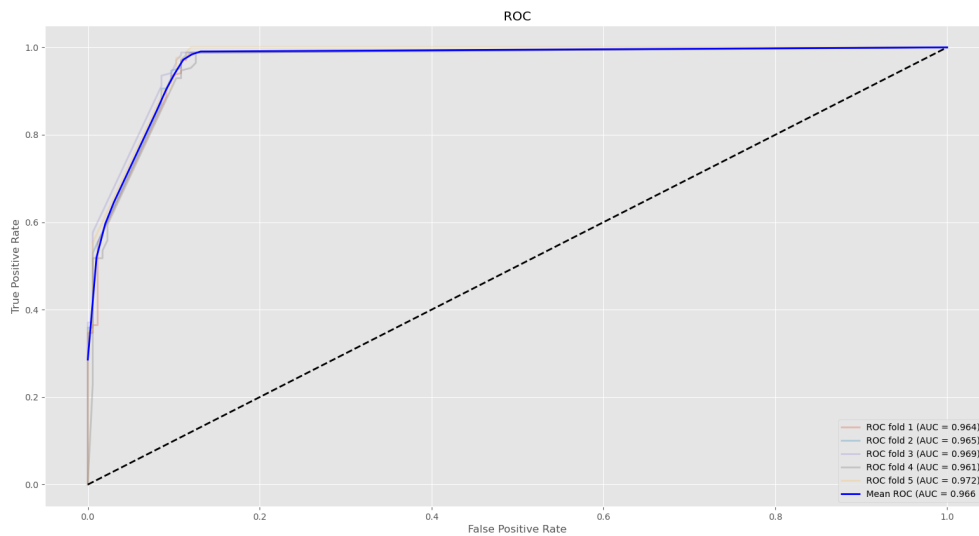
**Figure 5.24:** Tuning KNN for "Command char and length"

Using the KNN with 13 neighbors and Manhattan as distance measure we can see in Figure 5.25 the mean confusion matrix:



**Figure 5.25:** KNN confusion matrix for "Command char and length"

Calculating the AUC score for each fold we achieve a mean AUC of 0.966 for this feature set, which is slightly better than the first set we tested:

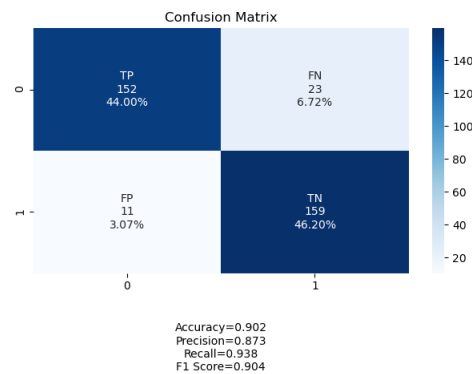


**Figure 5.26:** "Command char and length" AUC score for knn model

Tuning the decision tree classifier on the "Command char and length" set we got the following as the best tree parameters:

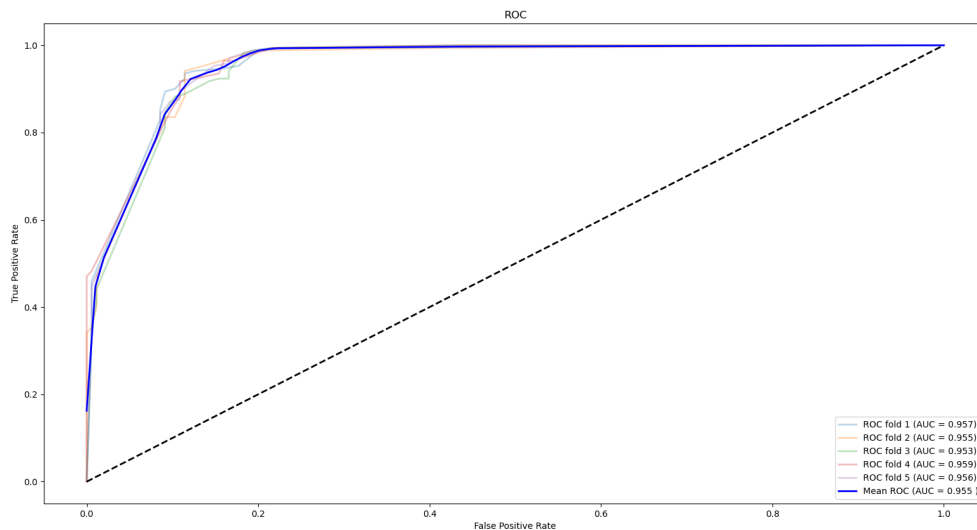
- criterion = gini
- max\_depth = 8
- min\_samples\_leaf = 14
- min\_samples\_split = 2

Confusion matrix achieved with this decision tree on "Command char and length" is presented in Figure 5.27:



**Figure 5.27:** Decision tree confusion matrix for "Command char and length"

AUC in each fold is presented in Figure 5.28, and we can see that the mean AUC is 0.955 with the decision tree which is 0.11 worse than the KNN:



**Figure 5.28:** "Command char and length" AUC score for Decision tree model

The third feature set we were to test was the one called "All variable features", which was the characters in the variables as well as their max, min and average length. This was inspired by the related work focusing on commands, which led us to exploring the variables as well. Tuning the KNN towards this feature set we see a top at 13 neighbours with euclidean distance before a small dip in the score for higher value of neighbors.

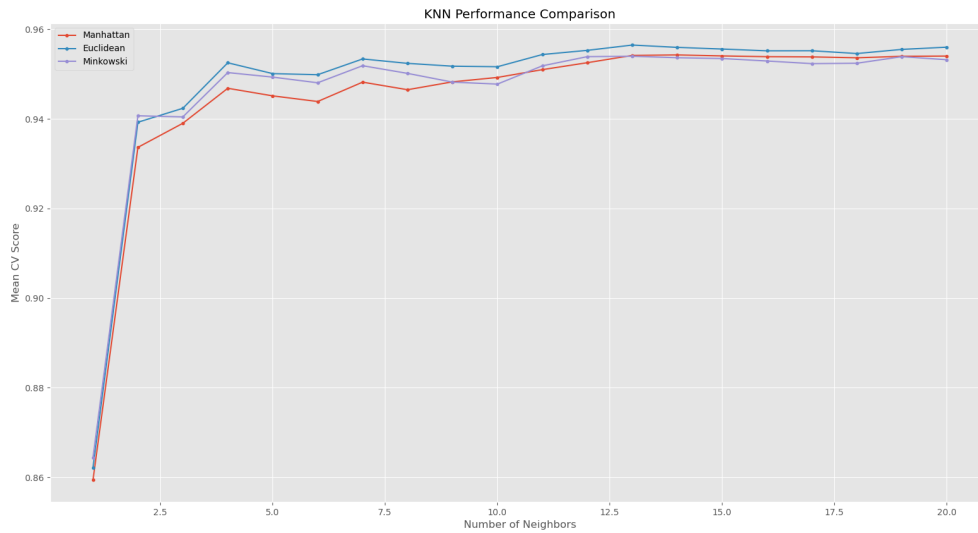


Figure 5.29: Tuning KNN for "All variable features"

Using the KNN with 13 neighbors and Euclidean as distance measure we can see in Figure 5.30 what the mean confusion matrix looks like:

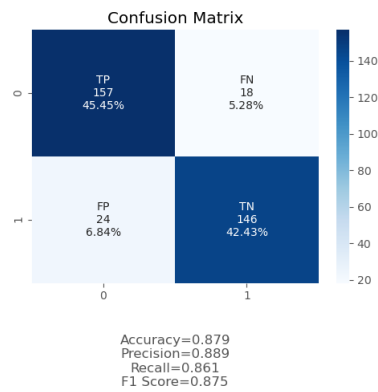


Figure 5.30: KNN confusion matrix for "All variable features"



Calculating the AUC score for each fold we achieve a mean AUC of 0.964 with this model and feature set, which slightly worse than the previous set:

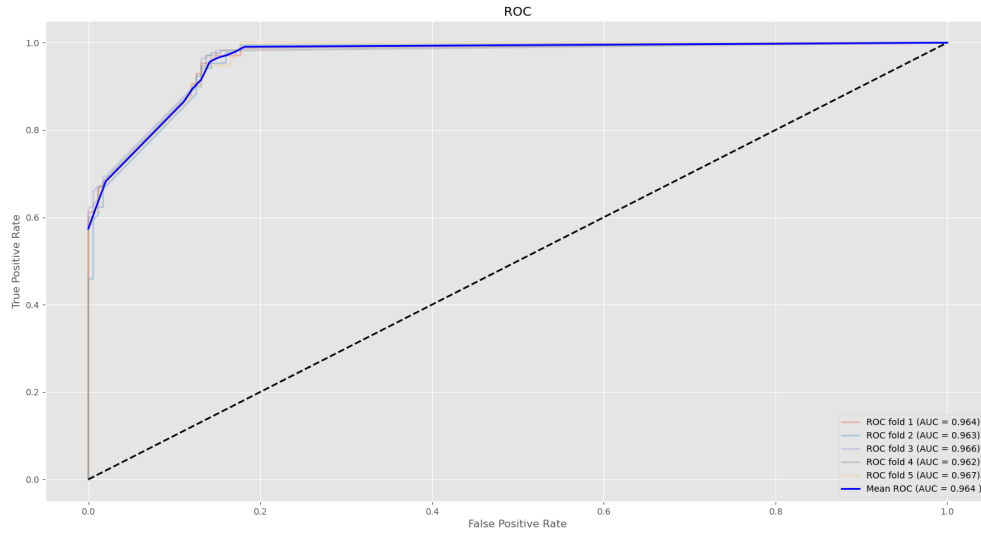


Figure 5.31: "All variable features" AUC score for knn model

Finding the potentially best decision tree for "All variable features" using the grid-search gave us the following tree parameters:

- criterion = gini
- max\_depth = 6
- min\_samples\_leaf = 1
- min\_samples\_split = 5

Confusion matrix achieved with this decision tree on "All variable features" is presented in Figure 5.32:

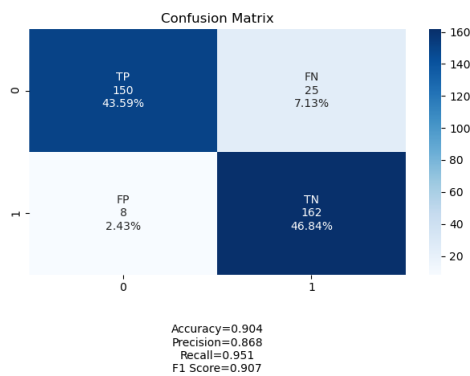
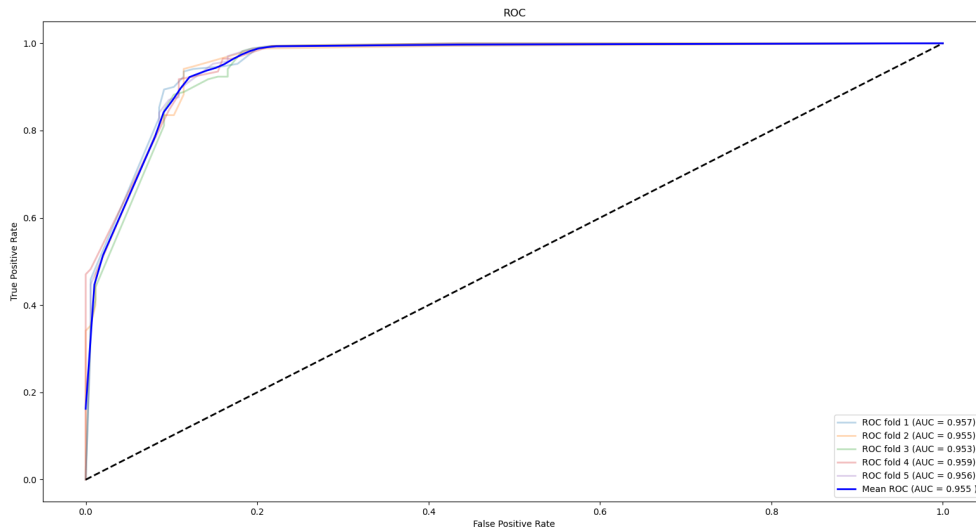


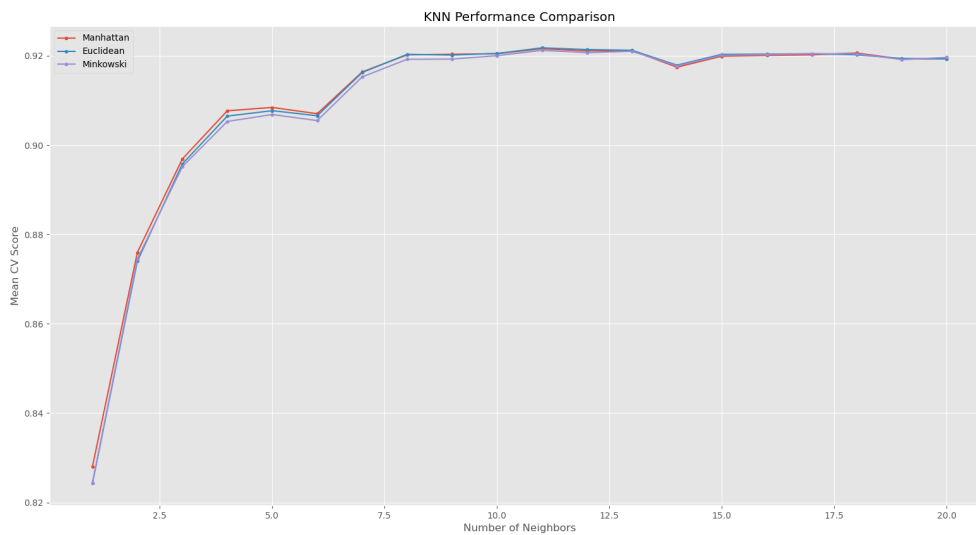
Figure 5.32: Decision tree confusion matrix for "All variable features"

AUC in each fold is presented in Figure 5.33, and we can see that the mean AUC is 0.959 with the decision tree which is 0.06 lower than what we achieved with the KNN:



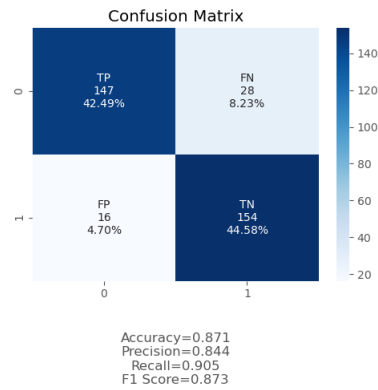
**Figure 5.33:** "All variable features" AUC score for Decision tree model

The fourth set we are testing is the "Verb check features", which is the verb check we created to see if the verbs used in commands are on Microsoft's approved list or not. This is where we are closing in on more advanced NLP tasks as we try to locate verbs and nouns and target PowerShell at one of the points where it is natural language like. In Figure 5.34 the top is located at 11 neighbors and using the Euclidean measure:



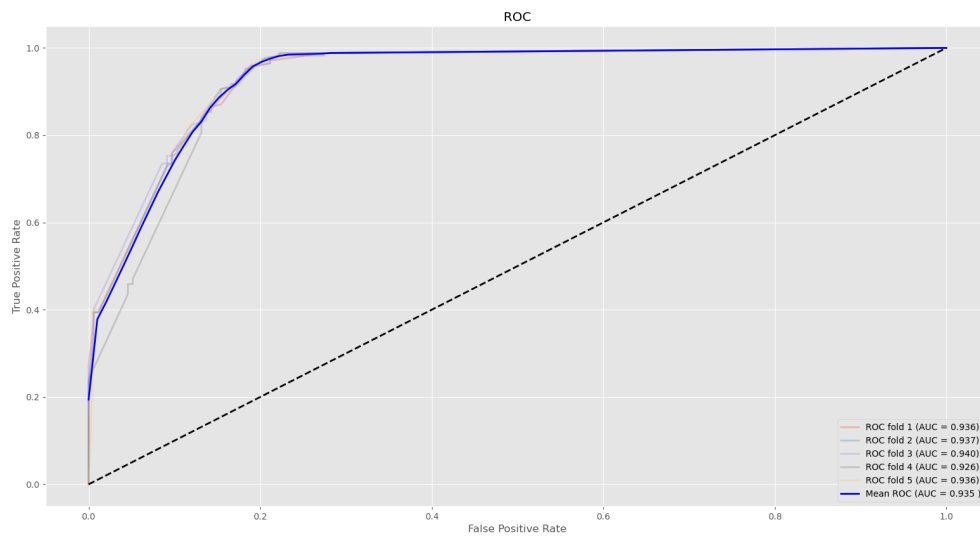
**Figure 5.34:** Tuning KNN for "Verb check features"

Implementing the KNN with 11 neighbors and Euclidean as distance measure we can see in Figure 5.35 how it performed:



**Figure 5.35:** KNN confusion matrix for "Verb check features"

Calculating the AUC score for each fold we see in Figure 5.36 that we achieve a mean AUC of 0.935 with this model and feature set which is slightly below what the other sets have achieved so far:

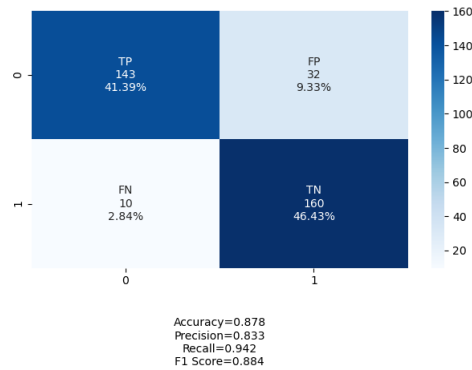


**Figure 5.36:** "Verb check features" AUC score for knn model

The decision tree we found to give the highest AUC score for the "Verb check features" was with the following parameters:

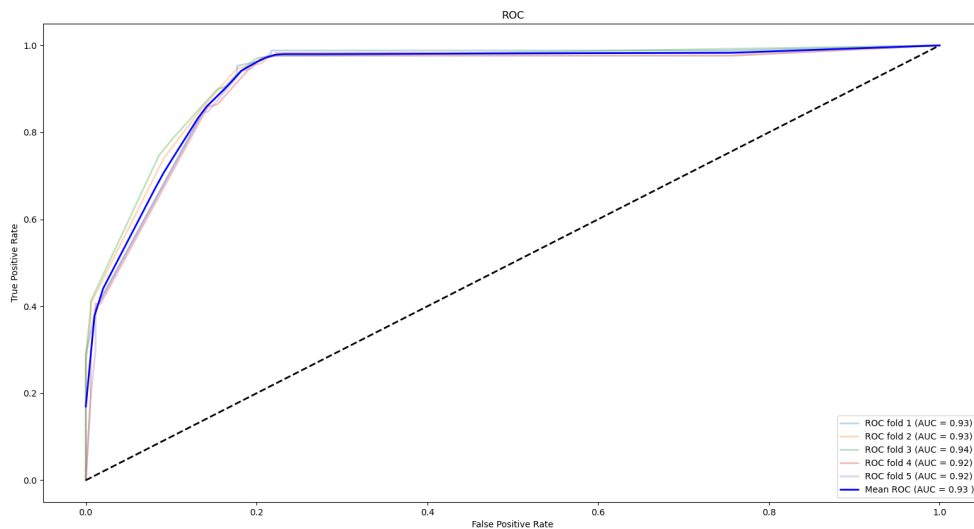
- criterion = gini
- max\_depth = 10
- min\_samples\_leaf = 3
- min\_samples\_split = 7

Utilising this decision tree model we achieved the following confusion matrix on "Verb check features":



**Figure 5.37:** Decision tree confusion matrix for "Verb check features"

We get an AUC mean score of 0.936 with the decision tree, which is the first time the decision tree has scored better than the KNN so far:



**Figure 5.38:** "Verb check features" AUC score for Decision tree model

The fifth set to be tested is the "Known commands" which is our wordcloud approach, previously explained, inspired by the related research using most frequent words. From this feature set we get the following result when tuning the KNN:

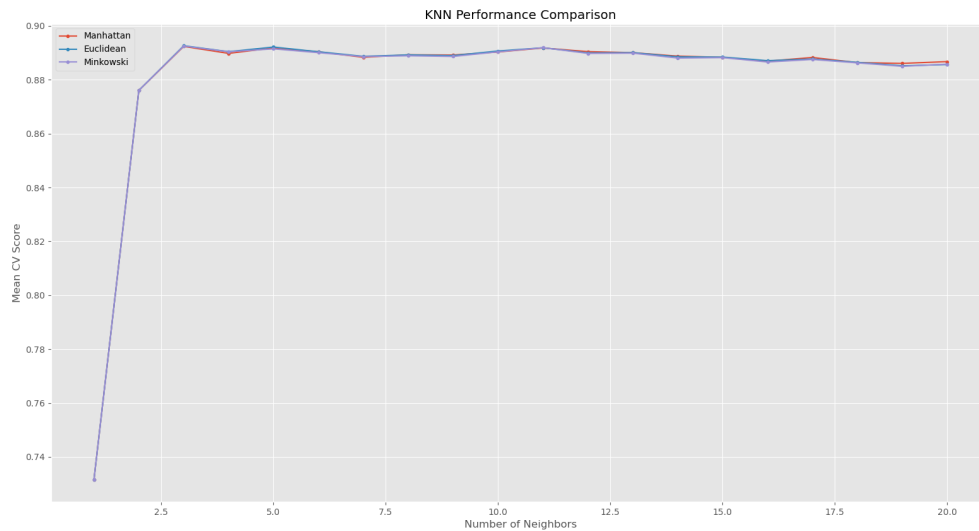


Figure 5.39: Tuning KNN for "Known commands"

It might be a little hard to spot on the figure, but the highest point is located at k neighbors equals 3, and we can see that the blue euclidean line is slightly above the others. Implementing the KNN with 3 neighbors and Euclidean as distance measure we get the following confusion matrix:

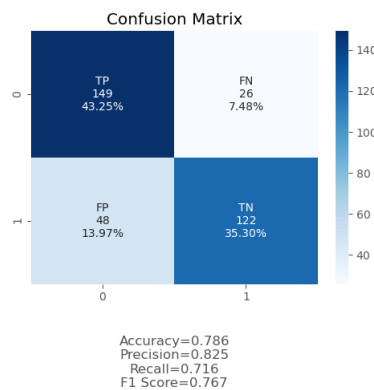


Figure 5.40: KNN confusion matrix for "Known commands"

Calculating the AUC score for the folds we can see that we get a mean AUC score of 0.907, and that is the lowest score of all the sets so far:

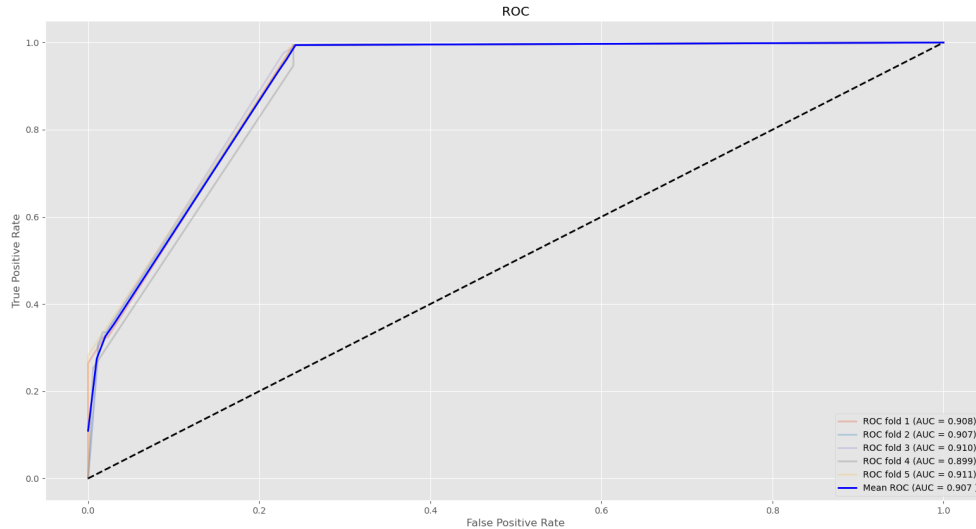


Figure 5.41: "Known commands" AUC score for knn model

Tuning the decision tree for the best parameters with the set "Known commands" we get the following parameters to give the best AUC score:

- criterion = gini
- max\_depth = 6
- min\_samples\_leaf = 3
- min\_samples\_split = 2

We achieved the following confusion matrix when implementing this decision tree on "Known commands":

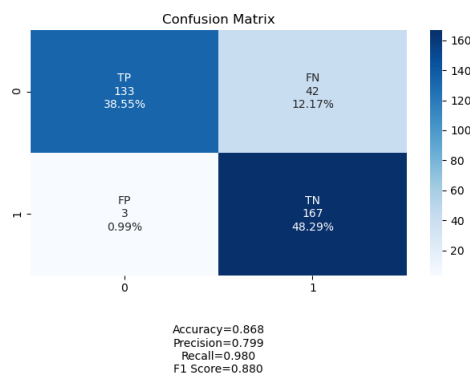
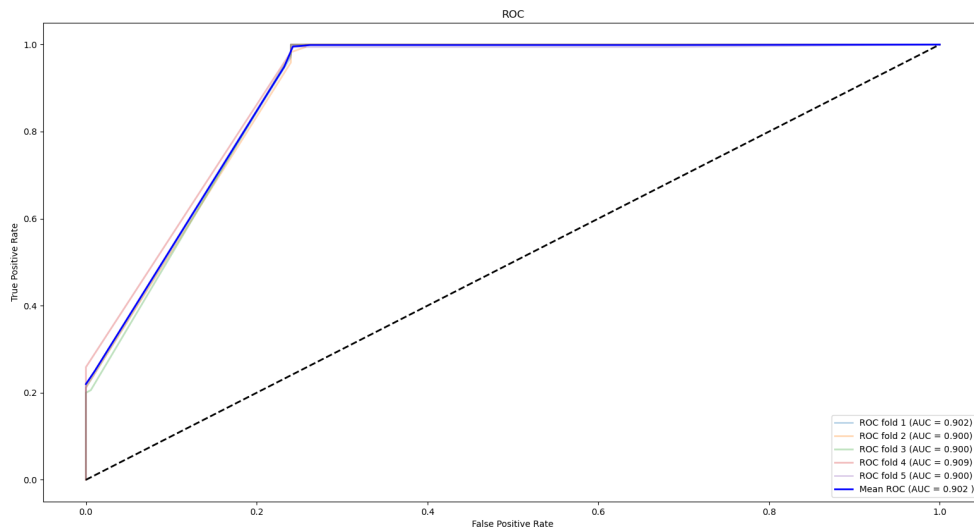


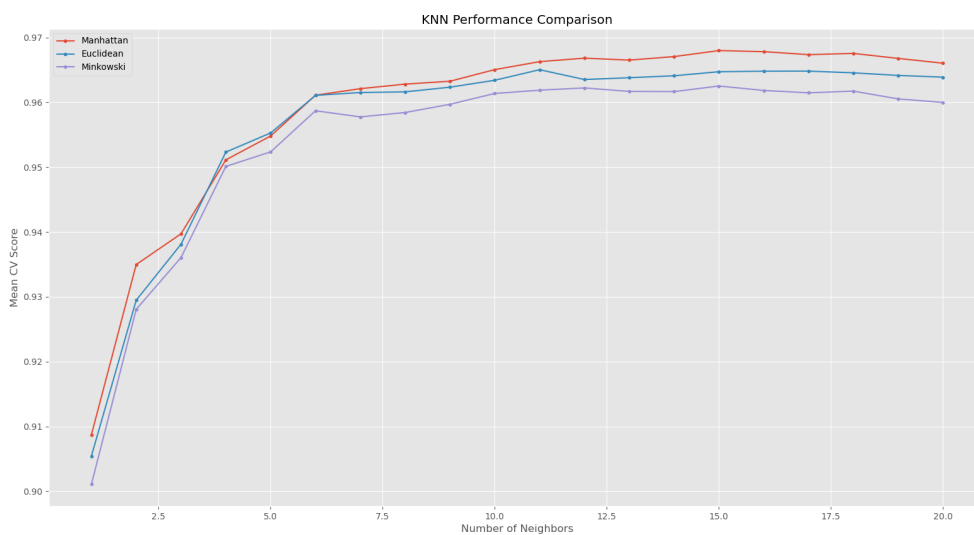
Figure 5.42: Decision tree confusion matrix for "Known commands"

From the folds we calculate the mean AUC score, which is 0.902 and that is 0.05 lower than the KNN:



**Figure 5.43:** "Known commands" AUC score for Decision tree model

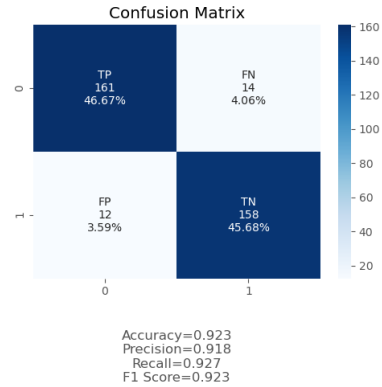
The last feature set we are going to test is all the features we so far have mentioned and tested separately. We named the set "All features" in Table 4.2, and the tuning of the KNN model on this set gave us the following parameters to use:



**Figure 5.44:** Tuning KNN for "All features"

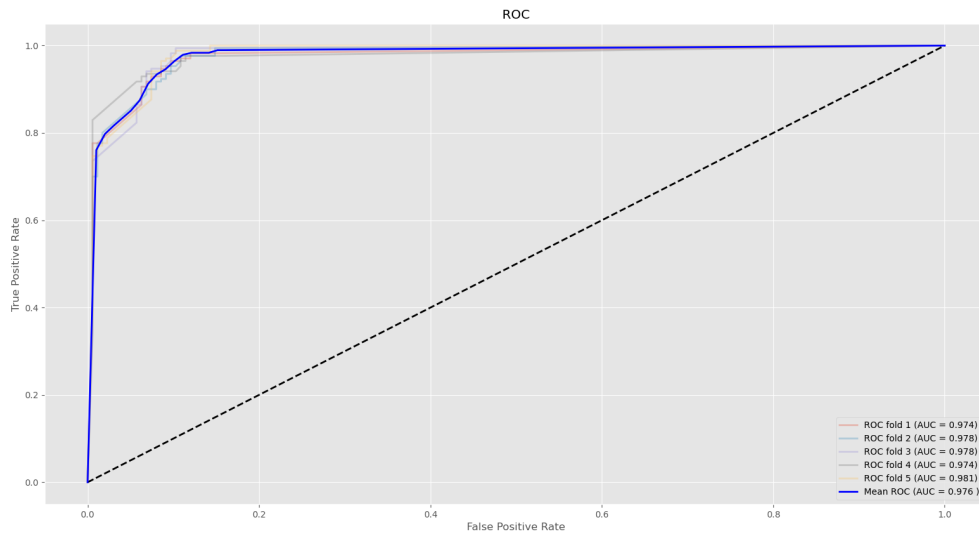
For all the features we can see that Figure 5.44 indicates that a number of 15 neighbors with the Manhattan measure are the parameters to achieve the best AUC score.

In Figure 5.45 we can see the confusion matrix we got when testing the KNN model on all the features:



**Figure 5.45:** KNN confusion matrix for "All features"

From the different folds we can calculate the mean AUC score, which we can see in Figure 5.46 were 0.976. That is so far the highest AUC score we have seen from any of the tested sets.



**Figure 5.46:** "All features" AUC score for knn model

The last decision tree we tune is on the set we called "All features", and we found the best parameters to be the following:

- criterion = gini
- max\_depth = 6
- min\_samples\_leaf = 10
- min\_samples\_split = 2

The decision tree model with those parameters gave us the following confusion



matrix:

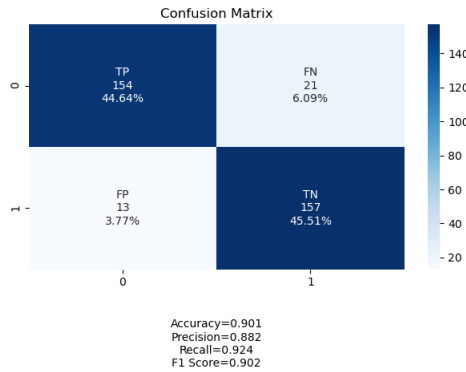


Figure 5.47: Decision tree confusion matrix for "All features"

Figure 5.48 presents how we used the different folds to calculate the mean AUC score, which was 0.965. That is 0.11 lower score than we achieved with the KNN:

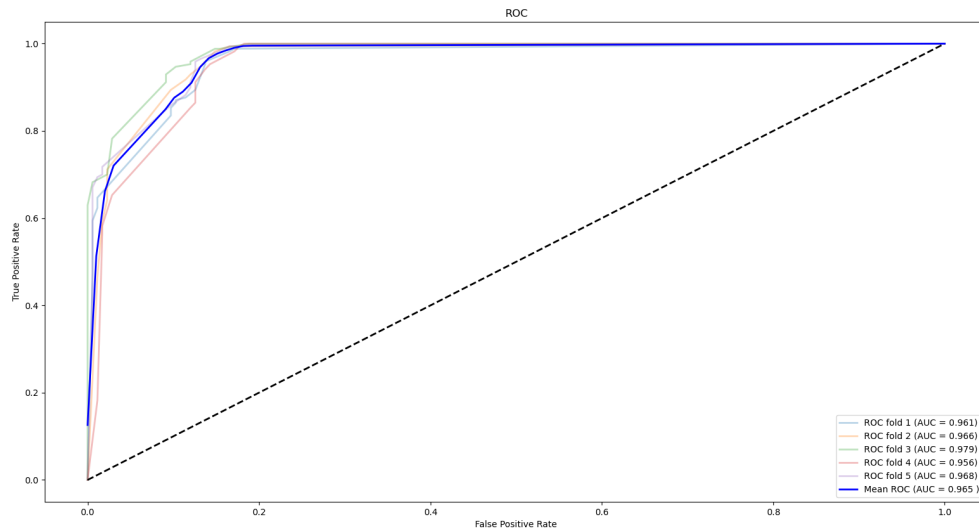


Figure 5.48: "All features" AUC score for Decision tree model

Going through all these figures might seem like a bit too much, but it might make it easier to follow the process and see where the numbers are coming from. This could potentially make it easier for other researchers who wants to continue the work in this thesis or tackle a similar classification problem. Figures might also make easier to understand for someone starting at the same knowledge level as we did, e.g., students. From our point of view it also makes it fast and simple to compare matrices and look for connections between the matrices and the diagrams.

So far we have not said much about the confusion matrices, and the reason is that we wanted to perform a easy to read comparison by creating a table. One

important aspect of these matrices are that they are also means of k-fold cross-validation, which means that even though the counts some times are double values we have rounded them to the closest integer for the figures. The different scores are presented as in the related research with three decimals. Creating a table when comparing the feature sets and model gave us Table 5.4. "DT" in the table means that the used model is the decision tree. "Acc" stands for accuracy:

Feature and (model)	AUC	Acc	Precision	Recall	F1
All command features (knn)	0.965	0.911	0.888	0.939	0.913
All command features (DT)	0.960	0.906	0.871	0.949	0.909
Command char and length (Knn)	0.966	0.920	0.897	0.946	0.921
Command char and length (DT)	0.955	0.902	0.873	0.938	0.904
All variable features (knn)	0.964	0.879	0.889	0.861	0.875
All variable features (DT)	0.959	0.904	0.868	0.951	0.907
Verb check features (knn)	0.935	0.871	0.844	0.905	0.873
Verb check features (DT)	0.936	0.875	0.834	0.932	0.880
Known commands (knn)	0.907	0.786	0.825	0.716	0.767
Known commands (DT)	0.902	0.868	0.799	<b>0.980</b>	0.880
All features (knn)	<b>0.976</b>	<b>0.923</b>	<b>0.918</b>	0.927	<b>0.923</b>
All features (DT)	0.965	0.901	0.882	0.924	0.902

Table 5.4: Classification scores

Table 5.5 shows the time used for training and testing the model on the related feature set with with stratified 5-fold cross-validation. To hopefully avoid the case of anything being stored in memory, we restart our system before running through the different models to get a reliable time used.

Feature and (model)	Time
All command features (knn)	0.38
All command features (DT)	0.14
Command char and length (Knn)	0.34
Command char and length (DT)	0.13
All variable features (knn)	0.31
All variable features (DT)	0.14
Verb check features (knn)	0.15
Verb check features (DT)	0.12
Known commands (knn)	0.21
Known commands (DT)	0.12
All features (knn)	0.53
All features (DT)	0.17

Table 5.5: Training and testing time

## Chapter 6

# Discussion

This chapter aims to discuss the results we presented in the previous chapter and try to draw lines and compare it to the related research. To give it a good structure we have divided the start of the chapter into sections answering our research questions before exploring possibilities for future work.

### **NLP technology for static analysis**

Our first research question was how NLP technology could be used to perform static analysis on malicious PowerShell scripts. This question was highly influenced by Tajiri[3] in 2020 stating almost no research had been done into how NLP technology could be used to classify malicious PowerShell. From the related work section we could see that there in the last couple of years has been a small amount of research done into this area where we saw how e.g., word- and character-embedding, word frequencies of commands and sentences of a set amount of tokens combined with deep learning already were tested and yielded good results for classifying malicious PowerShell. This led us towards identifying the possibility for using faster and more easily understandable models as a potential filter which had yielded positive results for Choi et al[20] when classifying PE-files and for Fass et al[22] when classifying JavaScript. We identified PSONeTools as a tool that could be used to parse and tokenize PowerShell code similarly to the how Fass et al. parsed and tokenized JavaScript. In the background chapter we described how we found the variables and commands to often be different when looking at the benign and malicious PowerShell scripts. But looking through a dozens of files isn't enough to safely draw any conclusion. PowerShell is as previously mentioned intended to use verb-noun names for commands, and by exploring the verbs we check whether or not the script uses syntactically correct commands according to Microsoft's naming convention. With the way we explored most frequent commands we try to some degree to understand the semantics. Letter frequencies and word lengths have been used for analysing languages for hundreds of years, and this statistical approach is another approach we tried to explore for the commands and variables. With these method we have identified a way to use NLP technology in order to classify malicious PowerShell. This leads to maybe the more important

question of how efficient this method is as we explore the achieved results to the performance of related research. We will see that our method performs slightly worse than the related research, which might indicate that our method is viable but could still use more work.

### **Classification performance**

Our second research question was how well our implemented solution would perform when trying to classify malicious PowerShell. In order to figure out how it performed we need to compare it to how the related research performed. Using the AUC score if provided seemed to be how most papers compared itself to other research, which is why we also will use this score. We tested our five different approaches by creating a feature set for each of them, and in the end a feature set with them all combined. Each set was tested towards a trained KNN model or decision tree. How each of these sets performed will be discussed when answering the next research question, but our best result was when using the feature set consisting of all the features combined and classified by the KNN model using the 15 nearest neighbors and the Manhattan distance. The achieved scores was as follows:

- AUC = 0.976
- Accuracy = 0.923
- Precision = 0.918
- Recall = 0.927
- F1 = 0.923
- Time = 0.53

How these scores are calculated were explained in the background chapter. The paper that was the closest related to our research was performed by Hendler et al.[18], where they achieved a AUC score of 0.995 at its best for powershell scripts collected within a time period of a few months. When testing the same models on scripts collected up to five months later they achieved a true positive rate, also known as sensitivity or recall, of 0.894 at its best. That means our method performed worse than what they achieved. If we compare it to how their model performed when testing the scripts collected at a later date we can see that our recall score of 0.927 is better. All the other related research we mentioned reported better results than our method has, but that does not mean our method or features are not useful. As previously mentioned we intended our method to be a fast classifier potentially to be used as a filter before more advanced deep learning methods could be applied. Scoring relatively close to the more complex and time consuming methods means that it could potentially help to implementing our method as a filter.

### **Best features**

The next research question was what might the best features for classifying malicious PowerShell when focusing on commands and variables. In order to answer

this we need to take a look at the individual feature sets we created. First we can start by looking at the set using all the command features, that being characters, signs, lengths, known commands and verb checks. Here we achieved a AUC score of 0.965. The second set called "command char and length" and we can see that the KNN performs a little better(0.966), but the decision tree performs a little worse. The difference with this set from the previous is that it doesn't contain the known commands features and verb check features. So far it might seem like the known commands and verb checks does not have a good impact. Moving on to the variable features we achieved a slightly lower AUC score(0.964) than with the set "Command char and length", and when comparing the two we confusion matrices we can see that they have the same amount of false negatives but the variables has way more false positives. For a business this could be important as they would like to have as few false positives as possible in order to keep the benign actions going. This might indicate that the commands are better than variables as indicators for whether the file is malicious or not.

The next set is the one based on our own experience with PowerShell and Microsoft's verb-noun naming convention for commands. An AUC score of 0.936 with the decision tree was the best achieved with only these features, which means that it is possible to use these features to classify the scripts, but it is not as useful as the character count and length approach previously tested for commands and variables. Once again will this approach not be able to differentiate scripts with no commands, but as it performs worse than the other command related sets it indicates that on its own it is a less accurate method. On the other hand we see that it is able to classify a good amount of the scripts correctly, so it indicates that there is a difference between malicious and benign scripts in regards to using "valid" verbs. From the confusion matrix we can see that it has more false negatives than false positives, which might indicate that a good amount of the malicious scripts do use the approved verbs.

The last individual feature set we tested was the "known commands" set which achieved the worst AUC score so far. This method also suffers from not being able to classify files with only variables, but we also suspect that since we used a very high threshold for sightings there might be a few files containing many of the same command while it not being present in others. From the confusion matrix we can actually see that it is better at classifying malicious scripts rather than benign, which might be when none of the commands being searched for exists it does not know how to differentiate the two types. Maybe a relative frequency approach would have been a better approach instead, in order to handle large scripts using the same command several times.

Finally we used all of the previous features sets together and achieved the best result(0.976) as described earlier in this chapter. When discussing these features we got a little worried that maybe it would perform better without the verb check and known commands features, but using all features yielded a better result. This indicates that even though the methods on their own had some struggles they proved to be of use when combined with the others. Comparing this confusion

matrix with the set "command char and length", which performed the best individually, we can see that the false positives has increased while the false negatives as been reduced. Meaning even though it performs better overall more benign files might be classified as malicious and might hinder benign activities in a company. The reason for this might be the fact that also using the variables helps correctly classify more malicious files, but since it had more false positives it makes it harder to identify the benign scripts.

We also claimed a desire to implement a fast and lightweight solution. The time needed for training and testing for our best model was 0.53 seconds. This is actually the time needed to perform 5-folds stratified crossvalidation, which means that it trains and tests 5 times before retrieving the mean result. Mimura et al.[19] needed 0.9 seconds and achieved a recall score of 0.990 and f1 score of 0.995. Comparing time used should be carefully compared as these implementations were performed on different systems, but our system used an i7-4770k cpu they used an i9-7900X. Our system had 16BG memory while theirs had 128. As previously stated it is risky to draw any conclusions based on time from very uneven systems like this, but it does indicate that our method is fast and lightweight. If we look at the decision tree on all the features it achieved slightly worse results, but was way faster which might actually be preferable. Keep in mind that this is a small dataset, which indicates that the KNN might struggle to keep up with the decision tree as the amount of scripts increases.

#### **Weaknesses of our solution**

The final research questing of ours was what the weaknesses of the implemented solution were. This approach is purely static analysis, which means that it is incredibly hard to ensure that our result is anything but indicators. The code is not being run, so even though the name of the commands and variables seems to be the same as in benign scripts they could be just using frequently used names. Another way to hide for this type of detection is hiding the malicious code within a larger benign script. This could potentially make the difference too small for the classifier to correctly identify it as malicious.

When comparing our research to the related work we can see that we find ourselves with one problem that occurred for many, which is the size of the dataset. We had the same amount of malicious files as some of them, but they chose to use way more benign samples. This is aspect we think our research did well in comparison when achieving almost a 1:1 ratio. But the fact that our dataset is very small in total compared to much of the related research it should be used as a small warning that our results might not perform as well on a larger dataset. It could also perform better, but noting the potential risk of a small dataset is at least important. Some of the assumptions made in this thesis could be taken as weaknesses as well. The first assumption we made was that our set of script files is a representation of how our method would be presented with files in a real life scenario and no further checks are being performed in order to ensure all files 100% are PowerShell files. Another assumption we made was the fact that we get to only have

non-obfuscated scripts, which means that a process for de-obfuscating scripts is assumed to be done. No further checks is being performed in this thesis in regards to obfuscated scripts. Since our assumption was that this is a representation of how we would be presented files in a real life scenario we did not use e.g, Virus-Total to double check the files.

### **future work**

The first and maybe most important aspect of future work is the size of the dataset. It is relatively small and in order to really ensure that our method is reliable we would need to gather more benign and malicious files. Hendler et al.[18] had a set of 100,000 scripts in comparison to our 1,725, which is quite a big difference. Achieving a set of that magnitude should definitely be a priority for future work. One of the problems with their dataset was the fact that it was very imbalanced in regards to benign:malicious ratio. Future work should in other words also focus on keeping the ratio between malicious and benign to be about the same ratio as achieved in this thesis. Mimura et al.[19] collected another test set containing scripts from up to 5 months later and achieved a worse result than when training and testing on scripts from the same time interval. This type of test should also be high priority for future work, and this is also one of the reasons why we split the features in to the different sets. We have a hypothesis that the known commands set would be the most affected by time as this specifically targets commands from one specific time and searches for them.

When we performed the feature selection for character frequency for commands and variables we used a very manual approach. This was performed before we learned how to use grid-search in an attempt to find the optimal parameters. The splits were also manually performed instead of using the k-fold approach, which shows our learning process during this thesis, but also makes it so that another future work maybe should revisit the feature ranking of characters. It also had the weakness of not being able to see all the possible connections because we implemented ranking based on individual performance. By using search methods for finding the best feature subsets it could be possible to obtain information not collected in this thesis. In order to find this subset future work could implement a subset search as explained by Dyrkolbotn et al.[26]. Our ranking method stated that the rare signs, e.g., Chinese characters, were of low importance but it would be interesting to see whether these rare signs occurred in the benign or malicious files. These characters could then potentially be used as signatures since they at least in our set very rarely occurred. We did not extract characters obtaining information about the case-sensitivity, and it would also be interesting to see if that would make any difference.

Without intentionally trying to we seemed to follow the same steps as Fass et al.[22] did when they were researching their JavaScript filter method. But in order for our method to really be a filter we need to use probabilities instead of pure classification. We have shown that our method achieves positive results when trying to perform a pure classification task, and the next step would be to create

thresholds for certainty in order to send the files we are uncertain about to further analysis, e.g., deep learning methods. The KNN is already based on probability as it takes a look at the  $k$  closest neighbours and classifies depending on which class is represented the most in the neighbours as well as the distance. Our model used the 15 closest neighbours which means we could need to set a threshold of how many of them to be of one class in order for us to be certain. As an example we could say we set the threshold at 12 neighbours of 1 class in order to be certain which would give us a threshold of 80%. Since the decision tree showed to be much faster than the KNN we can implement a random forest, just like Fass et al.[22] did, in order to get a set of decision trees. These decision trees would like previously give a binary class prediction, but since we now have several predictions we can, much like the KNN, set a threshold for how many trees to predict one class for us to be "sure". Just as an example we can say that we have 15 random trees, and we want 13 of them to classify a script with the same class in order to be certain about the class. The threshold would once again be 80% certainty.

We started on another method using the verbs of the commands in order to try to identify what type of action was being performed. So far our methods have not explored the behaviour of the scripts in a through fashion, which is why this approach to behaviour analysis was started. We have also seen that the previous verb check indicates that also several of the malicious scripts use this naming convention. First we identified the verbs and then used Microsoft's definition for what type of action the verb might indicate. They have divided the actions into common, communication, data, diagnostic, lifecycle and security. If the verb was not in the list we labeled it as "unknown". In Figure 6.1 we can see how the verbs in a benign file indicates what type of action is being performed. We can see that it actually contains a command using a verb not in the list:

```
{
  "VerbIndic": [
    "Common",
    "Common",
    "Unknown",
    "Common",
    "Data"
  ],
  "Name": "006597001301bbb1d8352887fb7b6df7cf1c21e34635fd0ae8016ddaaebd0816"
},
```

**Figure 6.1:** Verb action indicators from clean file

We can take a look at how it looks for a malicious file in Figure 6.2, and we can see that it also has some verbs indicating the actions aswell as one unknown verb:

Our plan was to use a Markov chain model in order to get the probability for jumping from one state to another. The different states would be the actions indicated by the verbs. This means that we would get two matrices for the probabilities for the transitions between the states. One for the malicious files and one for the be-



```
{
  "VerbIndic": [
    "Unknown",
    "Lifecycle",
    "Common",
    "Diagnostic"
  ],
  "Name": "240e3bd7209dc5151b3ead0285e29706dff5363b527d16ebcc2548c0450db819"
},
```

**Figure 6.2:** Verb action indicators from malicious file

nign files. When classifying the unknown files we would then get a matrix of its probabilities and we can use it to calculate the distance to the other two. The one that is closest would be the desired classification. For this method to be part of a potential filter we would need to set a threshold for the distance measure, e.g., euclidean distance, in order for us to classify those we are "sure" about and which ones to send on for further analysis. This extraction was created, but we were not efficient enough to properly implement a Markov chain model.

We pointed out that the related research of classifying PowerShell was highly focused on using deep learning, but it didn't utilize the latest within NLP models which is the transformers, also known as encoder-decoder technology. Just like Fass et al.[22] we tried to extract sentences using abstract trees, but we quickly realised that this task could be a thesis on its own. Using the BERT model[23] it would be possible to research a so far untested method for this classification problem. It could also potentially be used on our verb indicators as it, contrary to the markov chain model, is able to analyse a sequence in both directions. You could say markov chain reads from left to right, while the BERT model reads both ways to get even more information about a sequence.



## Chapter 7

# Conclusion

This chapter aims to draw conclusions based on the discussion from the previous chapter. Once again we have divided the chapter into sections regarding our research questions before ending with our thoughts about future work.

### **NLP technology for static analysis**

The goal of this thesis was to help gain knowledge about ways to classify malicious PowerShell, which is a relatively new area of research. Not much research has been done into using NLP technology for this type of classification, and the ones that did mainly focused on deep learning models. As a result of that we decided to research how other classification problems had helped deep learning models perform even better, which led us to our implemented solution. The solution follows the steps of feature extraction, feature selection and classification and evaluation. We focus on identifying commands and variables before extracting their character frequencies as well as min, max and average lengths. From the commands we also locate and extract the verbs and check them against Microsoft's approved verbs, and lastly we search for a few commands indicating malicious behaviour. This method achieved an AUC score of 0.976 and accuracy of 0.923 which is lower than most of the compared research, but as a potential filter we see a lot of potential. With these results we feel it is a fair conclusion to say that also within the area of malicious PowerShell classification a KNN or Decision tree could be used as a filter in order to reduce the amount of samples needed to be analysed by deep learning methods.

### **Classification performance**

As previously stated we concluded that our method has potential for being used as a filter based on the classification scores, but if it were to be used as a efficient and lightweight filter it needs to be fast in addition to good classification performance. Our result showed how our best performing model was a KNN using 0.53 seconds when training and testing the model using 5-fold stratified crossvalidation. We compared the KNN with a decision tree and we found the KNN to increase its time used much faster than the decision tree when using more and more features.

We can in other words conclude that our method is fast, but when increasing the dataset the decision tree will be much faster and the solution could be to implement a random forest model. This random forest model would also be needed in order to turn our method into a filter as explained in the discussion.

### **Best features**

Our results show that commands and variables achieve almost the same AUC score when focusing on character frequencies and lengths but there were enough difference for us to recommend one over the other. The "Command char and length" set achieved an AUC score of 0.966 while the "all variable features" achieved 0.964. This is an indication that commands might be the better indicator for whether a file is malicious or not. When using both the variables and commands we achieve the best result (0.976), which leads us to believe these sets compliment each other to better classify the malicious files but we also saw an increase in false positives using all the features. From the feature selection we can conclude that the ASCII characters as well as the numbers from 0 to 4 had the most individual importance for our classification problem

### **Weaknesses of our solution**

The most important weakness of this thesis is the size of our dataset. It is on the same level as other research stating this as a problem and much smaller than the largest datasets we have seen in related works. This problem affects all the steps taken in this thesis. Another weakness is the fact that our feature selection process used manually assigned stratified splits instead of k-fold crossvalidation. Our method is a static analysis approach, which means that it suffers from the weakness most static approaches do. The code is not being executed so it could be fooled by names trying to hide malicious commands and variables as benign, and our method should therefore be used as indicators. Malware creators could also hide malicious code within larger benign scripts in an attempt to fool our method.

### **future work**

Future work has immense amount of possibilities, and we highlighted some of these under the discussion chapter. The most important task for future work would be to increase the dataset and try to keep the balanced ratio between malicious and benign scripts. This could potentially have effect at all the steps we performed which means that even the character selection should be given another go, and with k-fold crossvalidation this time. Next we have our method for using the verbs as indicators for the action performed in a command that should be implemented like described under the discussion chapter. The next step would be to turn our knn model into a filter by applying a threshold, and a random forest model in order to turn the decision tree into a filter. More work should also be done into using the abstract syntax trees in order to create sentences used in deep learning. The deep learning to be used should be a transformer, e.g., BERT, in order to test the newest NLP deep learning models. That way we could be able to evaluate the

newest deep learning as well as a quantitative result of the impact the filter would have.



# Bibliography

- [1] *Understanding malware & other threats*, <https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/understanding-malware>, Accessed: 2020-10-17.
- [2] *Mcafee reports powershell malware attacks increased by 689% in 2020*, <https://www.techzine.eu/news/security/48632/mcafee-reports-powershell-malware-attacks-increased-by-689-in-2020/>, Accessed: 2020-10-17.
- [3] Y. Tajiri and M. Mimura, 'Detection of malicious powershell using word-level language models,' in *Advances in Information and Computer Security*, K. Aoki and A. Kanaoka, Eds., Cham: Springer International Publishing, 2020, pp. 39–56, ISBN: 978-3-030-58208-1.
- [4] C. Osborne, *Lockergoga: It's not all about the ransom*, <https://www.zdnet.com/article/industrial-malware-lockergoga-forces-victims-to-go-back-to-pen-and-paper/>, Accessed: 2020-10-17.
- [5] *A brief history of powershell*, [https://subscription.packtpub.com/book/application\\_development/9781784391492/1/ch01lvl1sec08/a-brief-history-of-powershell](https://subscription.packtpub.com/book/application_development/9781784391492/1/ch01lvl1sec08/a-brief-history-of-powershell), Accessed: 2021-03-20.
- [6] *Powershell versions*, [https://subscription.packtpub.com/book/application\\_development/9781784391492/1/ch01lvl1sec10/powershell-versions](https://subscription.packtpub.com/book/application_development/9781784391492/1/ch01lvl1sec10/powershell-versions), Accessed: 2021-03-20.
- [7] M. Sikorski and A. Honig, 'Practical malware analysis: The hands-on guide to dissecting malicious software,' in *Practical malware analysis*, No Starch Press, 2012.
- [8] S. Bragen, 'Malware detection through opcode sequence analysis using machine learning,' NTNU, 2015.
- [9] X. H. K. Griffin S. Schneider and T. Chiueh, 'Automatic generation of string signatures for malware detection,' Springer Berlin Heidelberg, 2009.
- [10] R. N. H. Yakura S. Shinozaki and Y. Oyama, 'Neural malware analysis with attention mechanism,' Elsevier Ltd, 2019.
- [11] X. N. M. Nguyen D. Nguyen and T. Quan, 'Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning,' Elsevier BV, 2018.

- [12] E. K. Y. Ki and H. Kim, 'A novel approach to detect malware based on api call sequence analysis,' SAGE Publications, 2015.
- [13] I. Kononenko and M. Kukar, 'Chapter 1 - introduction,' in *Machine Learning and Data Mining*, I. Kononenko and M. Kukar, Eds., Woodhead Publishing, 2007, pp. 1–36, ISBN: 978-1-904275-21-3. DOI: <https://doi.org/10.1533/9780857099440.1>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/B9781904275213500010>.
- [14] M. Balliauw and X. Decoster, 'Consuming and managing packages in a solution,' in *in Pro NuGet*, Berkeley, CA: Apress, 2013, pp. 13–46.
- [15] *Analyzing powershell threats using powershell debugging*, <https://darungrim.com/research/2019-10-01-analyzing-powershell-threats-using-powershell-debugging.html>, Accessed: 2021-03-25.
- [16] *Tokenizing powershell scripts - powershell.one*, <https://powershell.one/powershell-internals/parsing-and-tokenization/simple-tokenizer>, Accessed: 2021-03-25.
- [17] D. Hendler, S. Kels and A. Rubin, 'Detecting malicious powershell commands using deep neural networks,' in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS '18, Incheon, Republic of Korea: Association for Computing Machinery, 2018, ISBN: 9781450355766. DOI: 10.1145/3196494.3196511. [Online]. Available: <https://doi.org/10.1145/3196494.3196511>.
- [18] A. Rubin, S. Kels and D. Hendler, 'Detecting malicious powershell scripts using contextual embeddings,' May 2019.
- [19] M. Mimura and Y. Tajiri, 'Static detection of malicious powershell based on word embeddings,' 2021.
- [20] S. Choi, 'Combined knn classification and hierarchical similarity hash for fast malware detection,' 2020.
- [21] D. R. Patil and J. B. Patil, 'Malicious urls detection using decision tree classifiers and majority voting technique,' 2018.
- [22] A. Fass, M. Backes and B. Stock, 'Jstap: A static pre-filter for malicious javascript detection,' ser. ACSAC '19, San Juan, Puerto Rico, USA: Association for Computing Machinery, 2019, pp. 257–269, ISBN: 9781450376280. DOI: 10.1145/3359789.3359813. [Online]. Available: <https://doi.org/10.1145/3359789.3359813>.
- [23] M.-W. C. Devlin Jacob and K. Lee, 'Bert: Pre-training of deep bidirectional transformers for language understanding,' 2018.
- [24] *Virustotal*, <https://www.virustotal.com>, Accessed: 2021-04-05.
- [25] *Download virtualbox*, <https://www.virtualbox.org/wiki/Downloads>, Accessed: 2020-05-03.



- [26] G. O. Dyrkolbotn and E. Snekkenes, 'Electromagnetic side channel: A comparison of multi-class feature selection methods,' 2011.
- [27] *Scikit-learning*, <https://scikit-learn.org/stable/index.html>, Accessed: 2020-04-05.
- [28] *Sklearn.features.election.selectkbest*, [https://scikit-learn.org/stable/modules/generated/sklearn.feature\\_selection.SelectKBest.html](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html), Accessed: 2020-04-05.
- [29] *Sklearn.ensemble.extratreesclassifier*, <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html>, Accessed: 2020-04-05.
- [30] *Download and install windows powershell 5.1*, <https://docs.microsoft.com/en-us/skypeforbusiness/set-up-your-computer-for-windows-powershell/download-and-install-windows-powershell-5-1>, Accessed: 2020-05-03.
- [31] *Download the latest version for windows*, <https://www.python.org/downloads/>, Accessed: 2020-05-03.
- [32] *April 2021 (version 1.56)*, [https://code.visualstudio.com/updates/v1\\_56](https://code.visualstudio.com/updates/v1_56), Accessed: 2020-05-03.
- [33] *Installing pandas*, [https://pandas.pydata.org/pandas-docs/stable/getting\\_started/install.html](https://pandas.pydata.org/pandas-docs/stable/getting_started/install.html), Accessed: 2020-05-03.
- [34] *Installing scikit-learn*, <https://scikit-learn.org/stable/install.html>, Accessed: 2020-05-03.

