Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Iiro Kumpulainen

Adapting to Concept Drift in Malware Detection

Master's Thesis
Espoo, December 31, 2019

Supervisors:	Professor Aristides Gionis
Advisor:	Nikolaj Tatti D.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Iiro Kumpulainen		
Title: Adapting to Concept Drift in Malware Detection			
Date:	December 31, 2019	**Pages:**	55
Major:	Computer Science	**Code:**	SCI3042
Supervisors:	Professor Aristides Gionis		
Advisor:	Nikolaj Tatti D.Sc. (Tech.)		

This Master's thesis studies methods for detecting malware while adapting to how malware change over time. Malware detection methods are necessary to defend against cyber attacks, which could otherwise compromise information systems that are integral in the daily lives of everyone living in modern societies. Machine learning models have been shown to be effective in classifying files as either malicious or benign. However, malware and other software are constantly evolving, which makes classifiers unable to learn what malware in the future will be like. Therefore, methods are required for adapting to this concept drift in malware detection.

In this thesis, we compare three different methods called RETRAIN, THRESHOLD, and HYBRID that update the classifier to adapt to changes in the data over time. First, we create a classifier for detecting if a Windows portable executable file is malicious or benign. We use a dataset of 61466 files consisting of 19648 known malware and 41818 clean files. The classifier uses features that are extracted using static analysis without executing the files.

To choose which classifier to use in malware detection, we compare random forest, gradient boosted decision trees, neural network, and support vector machines. Gradient boosted decision trees achieve the highest accuracy and AUC, and using a random forest for selecting most important features reduces the training time of the model. This classifier achieves an accuracy of 98.7% when using cross-validation to evaluate the model. In contrast, when using temporally ordered files, the classifier has an accuracy of 96.7% showing the presence of concept drift in malware detection.

Each of the proposed RETRAIN, THRESHOLD, and HYBRID methods improve the accuracy of the classifier by adapting to concept drift. The best performing method is HYBRID, which is a combination of the two other methods. This method is able to correctly classify 98.5% of previously unseen files yielding a significant improvement in the malware detection capabilities of the classifier.

Keywords:	malware, malware detection, concept drift, machine learning
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Iiro Kumpulainen		
Työn nimi: Mukautuminen muutoksiin haittaohjelmien tunnistuksessa			
Päiväys:	31. joulukuuta 2019	**Sivumäärä:**	55
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvojat:	Professori Aristides Gionis		
Ohjaaja:	Tekniikan tohtori Nikolaj Tatti		

Tämä diplomityö tutkii menetelmiä haittaohjelmien tunnistamiseksi ja haittaohjelmissa tapahtuviin muutoksiin mukautumiseksi. Haittaohjelmien tunnistaminen on tärkeä tapa puolustautua kyberhyökkäyksiltä, jotka voisivat muutoin vaarantaa tärkeitä tietojärjestelmiä. Koneoppimismallit ovat tehokkaita luokittelemaan tiedostoja joko haitallisiksi tai vaarattomiksi. Haittaohjelmat ja muut ohjelmistot kehittyvät kuitenkin jatkuvasti, minkä vuoksi luokittelijat eivät pysty oppimaan, millaisia tulevaisuuden haittaohjelmat ovat. Tästä syystä tarvitaan menetelmiä haittaohjelmissa tapahtuviin muutoksiin mukautumiseksi.

Tässä opinnäytetyössä verrataan kolmea erilaista menetelmää, nimeltään RETRAIN, THRESHOLD ja HYBRID, jotka saavat luokittelijan mukautumaan ajan myötä tapahtuviin muutoksiin. Ensin työssä luodaan luokittelija, joka tunnistaa, onko Windowsin käyttöjärjestelmän suoritettava tiedosto haitallinen vai vaaraton. Luokitelijan kouluttamiseen käytetään 61466 tiedostoa, joista 19648 ovat tunnettuja haittaohjelmia ja 41818 on vaarattomia tiedostoja. Luokitteluohjelma käyttää ominaisuuksia, jotka on kerätty staattisella analyysillä tiedostoista ajamatta niitä.

Luokittelijan valitsemiseksi vertaamme menetelmiä random forest, gradient boosted decision trees, neural network ja support vector machines. Parhaan tarkkuuden saavuttaa gradient boosted decision trees -menetelmää käyttävä malli, ja kouluttamiseen kuluvaa aikaa voidaan nopeuttaa käyttämällä random forest -menetelmää valitsemaan tärkeimmät ominaisuudet tiedostojen luokittelemiseksi. Tämä luokitin saavuttaa 98.7% tarkkuuden ristiinvalidointia käytettäessä. Kun tiedostot järjestetään ajan mukaan, luokittimen tarkkuus on 96.7%, mikä näyttää, että haittaohjelmissa tapahtuu muutoksia ajan myötä.

Kukin ehdotetuista RETRAIN, THRESHOLD ja HYBRID menetelmistä parantaa luokittimen tarkkuutta. Paras menetelmä on HYBRID, joka on yhdistelmä kahdesta muusta menetelmästä. Tämä menetelmä luokittelee oikein 98.5% tiedostoista selvästi parantaen luokittimen haittaohjelmien tunnistuskykyjä.

Asiasanat:	haittaohjelma, tunnistaminen, muutos, koneoppiminen
Kieli:	Englanti

Acknowledgements

I wish to thank my advisor Nikolaj Tatti for all the advice and guidance in writing this thesis. I send my appreciation to my supervisor Aristides Gionis for supervising my work. I am grateful to the fellows at F-Secure who were supportive during my time there and who made it possible for me to work on this thesis.
Vaasa, December 31, 2019

Iiro Kumpulainen

Abbreviations and Acronyms

PE	Portable Executable
SHA-1	Secure Hashing Algorithm 1
API	application programming interface
JSON	JavaScript Object Notation
SVM	support vector machine
MLP	multilayer perceptron
ReLU	rectified linear unit
TP	true positive
FN	false negative
TN	true negative
FP	false positive
FPR	false positive rate
TPR	true positive rate
ROC	receiver operating characteristics
AUC	area under ROC curve
RF	random forest

Contents

Abbreviations and Acronyms **5**

1 Introduction **8**
- 1.1 Scope of the research 9
- 1.2 Structure of the thesis 9

2 Malware detection and concept drift **11**
- 2.1 Malware detection 11
- 2.2 Concept drift in malware detection 12

3 Dataset and feature extraction **13**
- 3.1 Included files 13
- 3.2 Feature extraction 14
  - 3.2.1 File identification 14
  - 3.2.2 Static file analysis 14
  - 3.2.3 Conversion to numeric features 16
- 3.3 Time distribution 19
  - 3.3.1 File timestamps 19
  - 3.3.2 First seen times 19

4 Methods for classifying malware **24**
- 4.1 Classification techniques 24
  - 4.1.1 Decision tree 24
  - 4.1.2 Random forest 25
  - 4.1.3 Gradient boosted decision trees 26
  - 4.1.4 Support vector machines 26
  - 4.1.5 Neural network 27
- 4.2 Data preprocessing methods 28
  - 4.2.1 Feature normalization 29
  - 4.2.2 Feature selection by importance in decision trees 30
- 4.3 Evaluation methods 30

4.3.1 Performance metrics 30
4.3.2 Cross-validation 31
4.4 Methods for concept drift adaptation 32
4.4.1 Model retraining 33
4.4.2 Model threshold adjusting 33
4.4.3 Hybrid method 35
4.4.4 Windowed variants 35

5 Experimental evaluation 38
5.1 Comparison of cross-validated classifiers 38
5.1.1 Experiment setup 38
5.1.2 Classifier comparison 39
5.2 Comparison of concept drift adaptation methods 42
5.2.1 Effect of temporally ordering data 42
5.2.2 Choice of retraining threshold 42
5.2.3 Method comparison results 43
5.2.4 Comparison with windowed methods 43

6 Conclusions and discussion 49

Chapter 1

Introduction

Malware, or malicious software, is any computer program that intentionally causes damage to a computer [33]. Nowadays, people are increasingly dependent on different information systems. Cyber attacks utilizing malware pose a significant threat to society, as demonstrated by the WannaCry ransomware attacks in 2017 infecting around 200 000 computers around the world and disrupting the health system in United Kingdom [5]. Furthermore, the amount of malware threats is constantly increasing, and the total number of different malware has exceeded 900 million in 2019 [2]. Thus, developing malware detection capabilities has a crucial influence on the entire modern civilization.

While it is theoretically impossible to create a malware detector that perfectly classifies every file as malicious or clean, in practice, we can create detectors that are adequate for real world purposes [12]. Traditional signature based malware detection methods are unable to detect previously unseen types of malware. For this reason, heuristic methods for malware detection have been developed. As the number of different malware is very high [2], crafting and updating comprehensive heuristic rules for malware detection by hand is extremely laborious. Hence, we use machine learning to automatically create these rules.

When applying machine learning to malware detection, we must consider that both malware and clean files change over time as new types of software are constantly being created [27]. This change in data distribution over time is called concept drift, and it makes it challenging to learn general and reliable malware detection rules. Gaining a better understanding of methods for adapting to concept drift enables us to create more accurate and versatile malware detection systems.

1.1 Scope of the research

Previous research has shown that using machine learning in static malware analysis is an effective way to detect malware [30]. However, it has been shown that traditional performance metrics are insufficient as concept drift must be taken into account in malware detection [27]. On the other hand, there is little research on what kind of methods should be used to adapt to concept drift in this domain.

In this Master's thesis, our goal is to create a malware detection system that adapts to changes in malware over time. We collect a dataset of 61466 portable executable files consisting of 19648 known malware and 41818 benign files for Windows operating systems. The files are selected from certification tests that evaluate the detection capabilities of commercial antivirus software. To extract features from the files, we use static analysis which means analyzing files without executing them.

We compare random forest, gradient boosted decision trees, neural network, and support vector machines to find which classifier performs best in malware detection. Our results show that gradient boosted decision trees are the most accurate classifier while random forest takes the least amount of training time. As our final classifier, we choose gradient boosted decision trees using features selected by a random forest to reduce the training time of the model.

After choosing the classifier, we evaluate its accuracy using cross-validation as well as using files that have been sorted by when they were first seen. These experiments show the presence of concept drift in our dataset of malware and clean files. For adapting to concept drift, we present three different methods and perform experiments to compare them. The first method retrains the model whenever the accuracy of the classifier falls below a set threshold. The second method adjusts the model threshold, which determines if an input file is classified as malware or as clean. The third method for concept drift adaptation is a hybrid method that combines the two other methods. Finally, we compare the results with using windowed variants, which only use the most recent files when adapting to concept drift.

1.2 Structure of the thesis

Chapter 2 gives a general overview of malware detection and concept drift. Chapter 3 describes the dataset and how features are extracted in order to employ machine learning. In Chapter 4, we introduce several machine learning and concept drift adaptation methods that are applied in this thesis.

Chapter 5 presents and analyzes the results of the experiments comparing the different methods for machine learning and for adapting to concept drift. Chapter 6 discusses the findings and concludes the thesis.

Chapter 2

Malware detection and concept drift

This chapter provides relevant background information for understanding later parts of the thesis. First, we give a brief overview of malware detection methods and previous research works related to our topic. Second, we introduce concept drift and discuss why it is relevant in malware detection.

2.1 Malware detection

Traditionally, malware are detected using signature based methods, which identify malware by searching files for patterns that are present in known malware [4]. This ensures that benign files are almost never detected as malware. However, these methods are unable to detect malware whose signatures are not already known.

In order to detect new malware, the files need to be further analyzed, which can be done using either static or dynamic analysis. Dynamic analysis means executing software in a controlled environment and analyzing their functionality. While behaviour based methods using dynamic analysis are useful in detecting new malware, analyzing files in this manner is computationally expensive. Additionally, behavioural methods are incapable of detecting malware that do not exhibit malicious behaviour during the analysis, such as malware that activate on a specific date [20].

In static analysis, files are analyzed without executing them. Instead, a file binary is examined by extracting static features such as syntactic library calls or n-grams, which are sequences of n consecutive bytes in the machine code of the file. While methods using static analysis of files are relatively fast, these methods may be susceptible to code obfuscation [38]. However, due to the time and resource intensive nature of dynamic analysis, this thesis is limited to only using static analysis of files.

Despite the potential weakness of being unable to reliably analyze obfuscated code, previous research has shown that classification methods using static analysis can be highly effective for malware detection. In 2000, Schultz et al. [50] used static analysis and data mining algorithms to automatically create rules that identify malware with 97.76% accuracy on a dataset of 4266 programs using cross-validation. These results were followed by dozens of studies using different classifiers and feature extraction methods for malware detection [20]. For instance, Kolter and Maloof [30] compared different classifiers using n-grams as features and concluded that boosted decision trees were the best classifier with an area under the ROC curve of 0.996. However, the results of these works cannot be directly compared with each other as they are using different datasets.

2.2 Concept drift in malware detection

In supervised machine learning for classification problems, a classifier is trained to predict a target variable using a labeled dataset of training examples. In this setting, concept drift refers to changing relation between input data and the target variable over time [19]. Concept drift may occur in dynamically changing environments, such as e-mail spam detection, where spammers may adapt their e-mails to avoid spam filters or a user may change their opinion on what kinds of e-mail should be regarded as spam. These changes in the data distribution make outdated classifiers unable to accurately classify recent samples, and methods for adapting to the concept drifts are required.

Previous research suggests that concept drift is also relevant in malware detection when using static analysis of files [27, 37]. Furthermore, prior studies have suggested methods for tracking concept drift in malware families [51] and notifying human analysts about identified concept drift in malware detection [26]. On the other hand, while different methods for automatically adapting to concept drift have been developed for many other domains [19], to the best of our knowledge there is lacking research on applying similar methods to malware detection.

Chapter 3

Dataset and feature extraction

This chapter describes the dataset that we used for the different experiments in this thesis. First, Section 3.1 describes the files that are included in the dataset and how they were selected. Section 3.2 then explains how we extract numeric features that describe the files in order to use the data for classification tasks. Finally, in Section 3.3, we discuss how the files are temporally ordered for concept drift analysis and show what the time distribution of files in our dataset looks like.

3.1 Included files

The dataset consists of 61466 files for Microsoft Windows operating systems in Portable Executable (PE) format. The PE format is used for files such as executable (EXE) and dynamic-link library (DLL) files, which include the code, data and resources that are required for running a software [40].

All files were obtained from F-Secure's file database. We selected the PE files by collecting only files that were used in tests performed by AV-Test,[1] an institution that evaluates the performance of commercial antivirus products. These tests by AV-Test were designed to find out how reliably different security products are able to detect real-world malware threats [1]. Therefore, we expect that experimental results on this dataset are meaningful and translate well to the real world.

In the tests performed by AV-Test, the antivirus software are tested on sets of recent and common known malware as well as on a separate large set of known benign files [1]. In contrast, in order to teach a classifier to detect malware and to analyze concept drift, we needed a dataset that contains

[1]`https://www.av-test.org/`

both malicious and clean files from the same time period. For malicious files, we collected 19648 malware that were used in tests from 2014 to 2018.

For clean files, we gathered 353626 benign files from AV-Test that were seen before the end of 2018. However, most of these files were other than portable executables, such as text or image files. As none of the malware in our dataset were of these types, it would be easy for a classifier to distinguish the non-executables as clean files by their file type. Therefore, we decided to only include the files that were portable executables, which resulted in 41818 clean files. Thus, the final dataset consists of 19648 known malware and 41818 known benign files for a total of 61466 files.

3.2 Feature extraction

As described in Section 2.1, this thesis is limited to analyzing files by using only static information without executing the files. In order to extract descriptive features of the PE files, we processed each file using a static file analysis service called TitaniumScale [42]. Finally, the analysis reports were parsed and transformed to numeric features for training machine learning models.

3.2.1 File identification

In order to uniquely identify the files, we used the cryptographic hash function SHA-1 (Secure Hash Algorithm 1): each file was represented by a 40 digits long hexadecimal number, which was computed by the SHA-1 hash function using the binary representation of the file as an input [16]. Using the SHA-1 hash allows us to specify, which file we are referring to without having to provide the entire binary code for that file.

The problem with identifying the files using the filename or another similar feature is that two different files may have the same values for these features, especially when a malware is trying to disguise itself as another file [8]. This problem does not occur when identifying files using SHA-1, because collisions between the SHA-1 hashes are extremely rare, which means that the SHA-1 identifiers for files are practically unique [36].

3.2.2 Static file analysis

We analyzed all 61466 files in our dataset by a cloud service called TitaniumScale, developed by the cyber security company ReversingLabs [42]. For each file, TitaniumScale performed a static analysis and created an analysis report, which included file headers, indicators of behaviour, imported application programming interfaces (API), as well as descriptions about resources and possibly other files embedded in the executable.

The analysis reports created by TitaniumScale were stored in Amazon Simple Storage Service (Amazon S3), a commercial cloud data storage service. We then used cluster-computing framework Apache Spark to collect the reports and transform the data into JavaScript Object Notation (JSON) format. Listing 3.1 shows an example of a file analysis report by TitaniumScale in JSON format.

```
"tc_report": [{
    "children": <omitted>,
    "classification": <omitted 12 lines showing file classification result>,
    "index": 0,
    "indicators": [{
        "category": 12,
        "description": "Checks operating system version.",
        "priority": 5
      }, {
        "category": 7,
        "description": "Takes screenshots.",
        "priority": 5
      }, {
        "category": 22,
        "description": "Deletes files.",
        "priority": 4
      },
      <omitted 30 indicators>
    ],
    "info": {
      "file": {
        "entropy": 6.429024772528842,
        "file_name": "ff0bf4b9ab2a291673b3ace52bef7c6596caecee",
        "file_path": "/scratch/ff0bf4b9ab2a291673b3ace52bef7c6596caecee",
        "file_subtype": "Exe",
        "file_type": "PE",
        "hashes": <omitted 16 lines showing hashes "imphash", "md5", "rha0", "sha1", and "sha256">,
        "size": 565258
      },
      "statistics": {
        "file_stats": <omitted 40 lines showing counts for different types of embedded files>
      },
      "validation": {
        "valid": true
      }
    },
    "metadata": {
      "application": {
        "capabilities": 626784511,
        "pe": {
          "dos_header": <omitted 19 lines showing "dos_header" values>,
          "file_header": {
            "characteristics": 271,
            "machine": 332,
            "number_of_sections": 4,
            "number_of_symbols": 0,
            "pointer_to_symbol_table": 0,
            "size_of_optional_headers": 224,
            "time_date_stamp": 1492923320,
            "time_date_stamp_decoded": "Sun Apr 23 04:55:20 2017"
          },
          "imports": [{
              "apis": ["RegOpenKeyExA", "CloseServiceHandle", "CreateServiceA", "DeleteService", "
                   OpenServiceA", "OpenSCManagerA", "ControlService", "StartServiceA", "
                   EnumServicesStatusA", "QueryServiceConfigA", "RegSetValueA", "RegCreateKeyA", "
                   GetFileSecurityA", "SetFileSecurityA", "RegDeleteValueA", "RegSetValueExA", "
                   RegQueryValueExA", "ChangeServiceConfig2A", "RegCreateKeyExA", "RegDeleteKeyA", "
                   RegOpenKeyA", "RegEnumKeyA", "RegCloseKey", "RegQueryValueA", "
                   ChangeServiceConfigA", "QueryServiceConfig2A"],
              "name": "ADVAPI32.dll"
            }, {
              "apis": ["ImageList_ReplaceIcon", "0x0011", "ImageList_Destroy", "ImageList_Create"],
              "name": "COMCTL32.dll"
58             },
59             <omitted 10 imports>
60           ],
61           <omitted 98 lines describing "optional_header" and "overlay">
62           "resources": [{
63               "code_page": 0,
64               "hashes": <omitted 10 lines showing hashes "md5", "sha1", and "sha256">
65               "language_id": 2052,
66               "language_id_name": "Chinese - People's Republic of China",
67               "name": "1",
68               "offset": 375696,
69               "size": 1128,
70               "type": "RT_ICON"
71             }, {
72               "code_page": 0,
73               "hashes": <omitted 10 lines showing hashes "md5", "sha1", and "sha256">
74               "language_id": 2052,
75               "language_id_name": "Chinese - People's Republic of China",
76               "name": "2",
77               "offset": 376824,
78               "size": 2440,
79               "type": "RT_ICON"
80             },
81             <omitted 1008 lines describing 55 other resources>
82           ],
83           <omitted 154 lines describing "rich_header", "sections", and "version_info">
84         }
85       }
86     },
87     "story": "This file (SHA1: ff0bf4b9ab2a291673b3ace52bef7c6596caecee) is a 32-bit portable
           executable application. The application uses the Windows graphical user interface (GUI)
           subsystem, while the language used is Chinese from People's Republic of China. According to
           version information, this is MySerManag \u5e94\u7528\u7a0b\u5e8f (MySerManag.EXE). Appended
           data was detected at the file's end. Its length is smaller than the size of the image. This
           application has access to device configuration, monitoring and running processes, has
           security related capabilities and uses undocumented APIs. There are 36 extracted files.",
88     "tags": ["antidebugging", "arch-x86", "capability-execution", "capability-filesystem", "
           capability-security", "capability-undocumented", "desktop", "gui", "indicator-execution", "
           indicator-file", "indicator-monitor", "indicator-registry", "indicator-search", "indicator-
           settings", "overlay", "privacy-intrusion", "rich-header", "version-info"]
89   },
90   <omitted 2547 lines describing 36 resource files embedded in the executable>
91 ]
```

Listing 3.1: TitaniumScale analysis report for the file with SHA-1 hash value ff0bf4b9ab2a291673b3ace52bef7c6596caecee. This file is a malware disguised as a Microsoft file called MySerManag.exe. The analysis shows that the executable has indicators of taking screenshots and deleting files, among other behaviour. The report also contains metadata about the executable as well as information about embedded files.

3.2.3 Conversion to numeric features

In order to use the data to train a classifier, the data from TitaniumScale had to be transformed to a suitable format. We did this by using the structure of the TitaniumScale analysis report to create features for all the available values. For example, for the entropy value on Line 22 in Listing 3.1, we created a feature with the name "/info/file/entropy/", and for this file the feature has the value 6.429024772528842. On the other hand, for string features such as the tags on Line 88, we created features "/tags/'antidebugging'/", "/tags/'arch-x86'", and so on, each having value 1 indicating that this file has that tag. Correspondingly, all files that do not have particular tags were then given value 0 for the respective features.

In some parts of the TitaniumScale reports, the attributes were grouped in sets, such as the values for indicator "category", "description", and "priority" in Lines 5–17 in Listing 3.1. Simply following the structure of the report would yield features such as "/indicator/category", whose values would be ambiguous as different indicators may have different categories. In order to avoid this, the values in these kind of sets were grouped by a value such as "name" or "description", which defines which set the values belong to. For example, the values in Lines 6–8 were grouped using the "description" attribute and thus represented as a feature "/indicator/'Checks operating system version.'" with value 1, as well as features "/indicator/'Checks operating system version.'/category" and "/indicator/'Checks operating system version.'/priority" with values 12 and 5, respectively. This grouping allows us to match the category and priority values with the correct indicators. As an exception, sets of attributes were not grouped in this manner in the resources section of a file report, shown on Lines 62–82. That is because resources were named using numbers such as "1" and "2" and often completely different resources in different files have the same value of "name" attribute. In contrast, when different TitaniumScale reports have the same descriptions for indicators or names for imported libraries, they are referring to the same indicator or library.

For the resources section on Lines 62–82 and for other cases where another file having a similar set does not mean they are referring to the same object, we created features which describe statistics about the values in these sets. For example, for the resource sizes on Lines 69 and 78, instead describing the sizes using a feature such as "/metadata/application/pe/resources/size" which should have a different value for each different resource, we created features for describing the minimum, maximum and mean sizes of the different resources as well as a count of how many resources there are in the file. For string features in these sets, we created a feature describing how many times this string occurs in the file. For instance, for the resource type on Lines 70 and 79, we created a feature "/metadata/application/pe/resources/type/'RT_ICON'", whose value indicates how many resources of the type "RT_ICON" there are in this file.

Many of the portable executables contain embedded files, such as other PE files or resource files used by the executable. TitaniumScale recursively processes all embedded files and includes the extracted information in the analysis report. If the report contained descriptions about other PE files, we dealt with the hierarchical structure by creating features for the each of the values of the embedded PE files in the same manner as for the main PE file but by prepending the feature names with "/child". Thus, for instance the entropy of a lone embedded PE file was stored as a feature "/child/info/-file/entropy". In the case of multiple embedded PE files, we again computed the count as well as minimum, maximum and mean values for the entropies of the different embedded PE files. We did not create new features to describe values in embedded files other than PE files, because most of the information about embedded resource files is already included in the resources section of the main PE file.

The total number of different features created by this process was around 50 million, when collecting features from half of the files in our dataset, and around 70 million while using 80% of the files. Instead of collecting new features from all of the files, the dataset was divided in each experiment into two sets: a training set, which was used for training a model, and a test set, which was used to evaluate the performance of the trained model. The model is not able to learn how to use features that are only present in the testing set since they are not relevant during training. Thus, the total number of different features for a particular experiment only depended on the files in the training set.

Most of these obtained features only appeared in very few files or were unique, such as the features indicating particular filenames or hashes. As these features had a value of zero for almost all files, they provided little to no information to a classifier. In order to make the total number of features more manageable, we limited to only using features which appeared in at least 1% of the files that were used to train the classifier for that experiment.

In addition, the creation timestamps of the files in our dataset are biased as shown in Section 3.3.2, because AV-Test uses different procedures for selecting malware and for clean files [1], and most of the clean files in the dataset were from a different source. Looking at the timestamps of the files in our dataset may thus help in distinguishing which files are malware, whereas in the real world we believe that the timestamps are not useful in this manner for detecting malware. Therefore, the timestamps of the files were not included as a feature in order to prevent this data leakage.

Furthermore, we discarded the features related to the file classification results by TitaniumScale, because we do not want our classifier to rely on another classifier. This is especially important since this thesis includes experiments where our model is restrained to only having information prior to a certain point in time, while the classifier in TitaniumScale may be using information from after that time period.

Finally, we removed any features that have the same value for all files. In total, these reductions decreased the number of features to around 6000-6500 both when using half of the files and when using 80% of the files in our dataset for training. The exact number of features varied depending on the particular set of training files for an experiment.

3.3 Time distribution

As this thesis examines how time factors into malware detection, it is important to have a reliable way to date each file. We show that the creation timestamp included in the file metadata is not always accurate. Therefore, instead of the file timestamps, we use the times when the file was first seen by F-Secure.

3.3.1 File timestamps

The Portable Executable file format for Windows operating systems includes a timestamp, which indicates when the file was created [47]. Many previous works on malware detection use the embedded file timestamps when sorting the files by time [48, 51]. Nevertheless, there is no guarantee for the accuracy of these timestamps as they may be arbitrarily set to any value by modifying the executable [9].

In our dataset, many of the files have incorrect timestamps as shown in the histogram of timestamps in Figure 3.1. Thousands of files, both malware and clean files, have clearly forged timestamps indicating that they were created in the future or unrealistically far in the past. In addition, not all portable executable files adhere to the PE format specification, as our dataset contains 22 files that do not include a timestamp. Thus, these timestamps are not reliable for temporal analysis of the files. Saxe and Berlin [48] deal with this problem by excluding all files with blatantly tampered timestamps from their dataset. However, this limits the dataset, and the remaining files may still have incorrect timestamps.

3.3.2 First seen times

A solution to the problem of unreliable file timestamps is to use the dates when the files were first discovered [27]. We apply this solution by using the date a file was first analyzed by F-Secure to assign a time to each file. A histogram of the times assigned for the files is shown in Figure 3.2. This results in a reliable way to temporally order the files, which means that we may train a machine learning model to detect malware without using information about files that are only seen later in time.

In our dataset, malware and clean files have differences between the time distributions, which is caused by having separate methods for collecting malicious and clean files. Figure 3.3 shows how the ratio between malicious and clean samples changes over time for the files in our dataset. Since a real-world malware detector likely would not have such drastic shifts in receiving malware and clean files, we need to be careful in our experiments so that the results are not skewed due these differences between our dataset and reality.

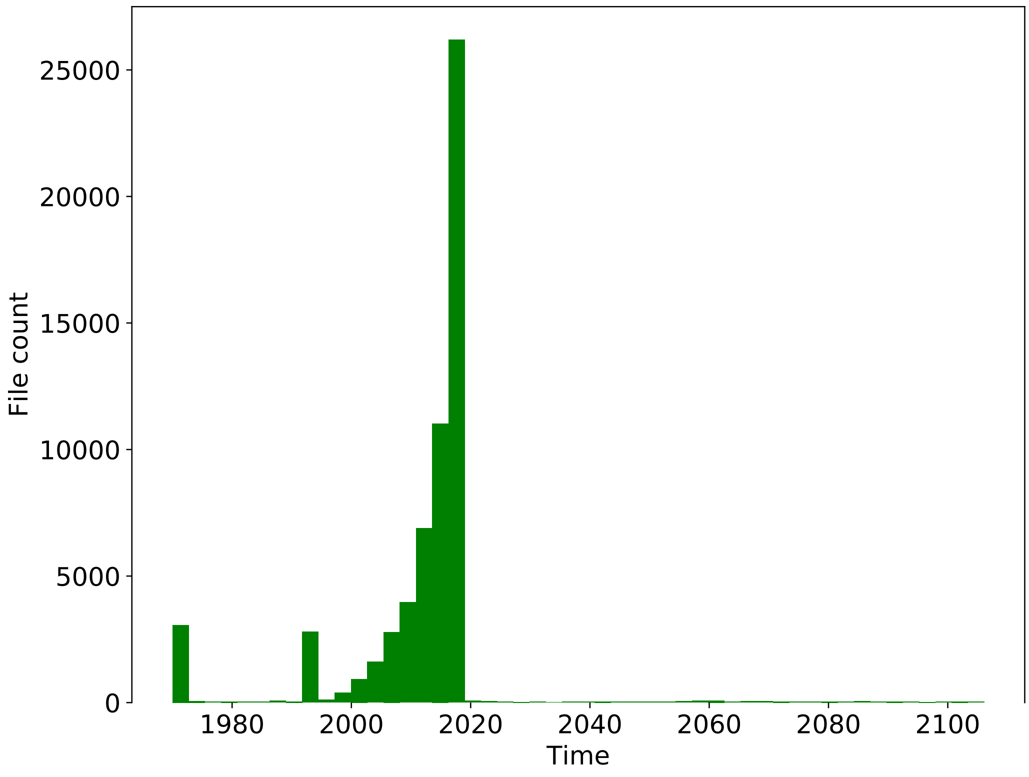

Figure 3.1: Histogram of timestamp dates embedded in the files in our dataset. There are many files whose timestamp incorrectly indicates that they were created before the 21st century or in the future.

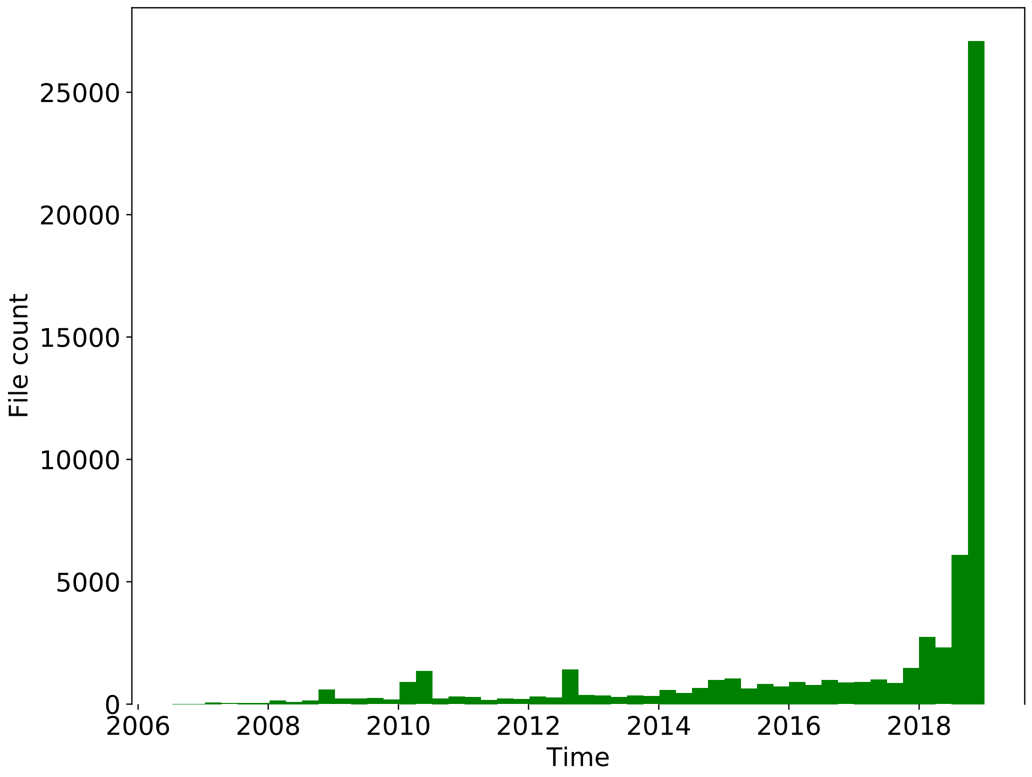

Figure 3.2: Histogram of dates when the files in our dataset were first seen by F-Secure. The first few files date back to 2006 while the vast majority of files are from 2018.

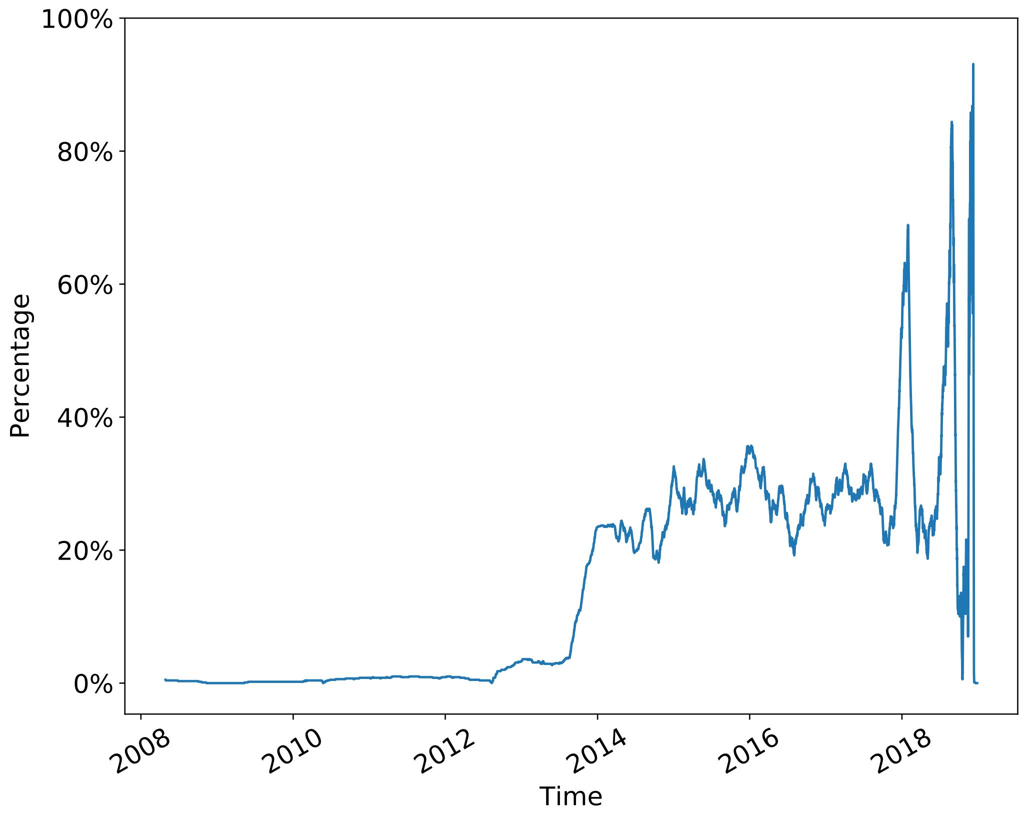

Figure 3.3: Percentage of malware in a sliding window of 1000 consecutive samples. The earliest files are almost all clean files, whereas from late 2013 to late 2017, the proportion of malware hovers between 20% and 40%. Towards the end of 2017 and in 2018, the files alternate between mostly malware and mostly clean files.

Chapter 4

Methods for classifying malware

This chapter describes different machine learning methods that are used in this thesis for creating a classifier for malware detection. First, we introduce different classification techniques as well as methods for preprocessing the data before using it to train a classifier. Then, we explain how the performance of the classifiers is evaluated. Finally, we present a few different methods for adjusting the machine learning model to changes in the data.

4.1 Classification techniques

In order to classify files as either malware or clean, we need to train the classifier with a set of samples with known labels. The classifier tries to gain insight into the data and accurately predict the label for new samples. There are many different classification algorithms for achieving this purpose, and the best choice of algorithm always depends on the problem at hand, see [32], for example. This section introduces common classification techniques, whose performances in malware detection are compared in this thesis.

4.1.1 Decision tree

A decision tree is a tree graph that models decision making by breaking complex decisions into a combination of multiple simpler decisions [44]. In classification, internal nodes of the decision tree represent tests for attributes of the input, edges represent the results of these tests, and each leaf node represents a class as the resulting classification [43]. Figure 4.1 shows an example of a simple decision tree classifier.

Decision tree classifiers are automatically constructed from the attributes and class labels in a given set of training data. When the amount of data

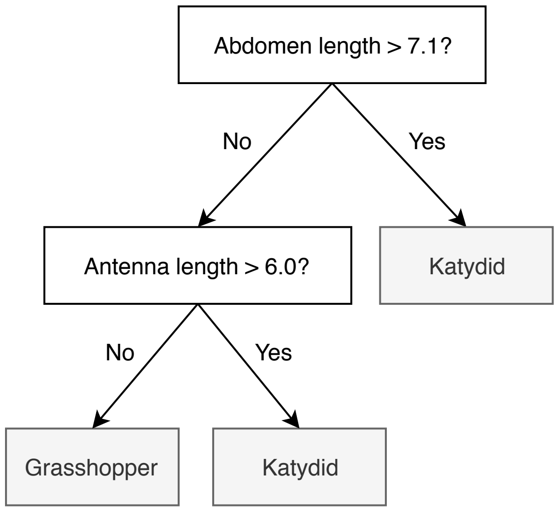

Figure 4.1: A decision tree that classifies an insect as a grasshopper or a katydid based on the lengths of the insect's abdomen and antenna [52].

is large, it is not feasible to compute an optimal decision tree for classifying a set of samples [21]. Consequently, most existing methods for constructing decision trees use heuristic algorithms that select greedily, which attributes to consider at each step [43].

Once trained, decision trees are fast at classifying samples. Additionally, the functionality of a decision tree is easier to interpret than for many other machine learning methods. However, decision trees are prone to overfitting to the training data, which means that the decision tree does not generalize for samples outside of the training set. In general, other machine learning methods tend to achieve better accuracy than plain decision trees. [32].

4.1.2 Random forest

Random forest is a machine learning model that uses bagging of decision trees, which means that multiple independent decision trees are combined to create one model. In classification, each decision tree casts a vote by classifying the input, and the random forest model then chooses class with the highest number of votes. Taking the majority vote of multiple identically distributed trees decreases the variance of the prediction, resulting in better accuracy than individual decision trees. [6].

The decision trees in the random forest are trained using randomly selected subsets of the training samples and each tree only considers some subset of the available features in the data. This reduces the correlation between individual trees and further decreases the variance of the random forest ensemble. [22]. Random forests are relatively simple and fast to train while they also achieve good accuracy in many practical problems [6].

4.1.3 Gradient boosted decision trees

Gradient boosting is another method for combining multiple simpler models, such as small decision trees, to create a more powerful predictor. Gradient boosted decision trees create a classifier by sequentially adding new decision trees, where each tree tries to correct the mistakes of the previous model [22]: The classifier training starts with an initial predictor that simply gives the same probability for each class. Then, at each iteration of the algorithm, a new decision tree is trained for each class to predict how the probability given by the current predictor differs from the true value of 0 or 1 that indicates whether a sample belongs to that particular class. This decision tree is then combined with the predictor to decrease the errors in the predicted class probabilities, thus slightly improving the predictor in each iteration. [34]. After a fixed number of iterations, the algorithm finishes, and the resulting model classifies inputs by predicting the probabilities for each class and choosing the class with the highest probability [22].

Unlike in a random forest, the individual decision trees are not allowed to grow freely but are limited to having a small fixed size in order to improve the performance of the model [18]. In general, gradient boosted decision trees are very effective and achieve state-of-the-art performance in many different machine learning tasks [11].

4.1.4 Support vector machines

Support vector machines (SVMs) are a popular machine learning method, which use a hyperplane to divide the data into two classes. The data samples are considered as points in a multidimensional space, where each data feature defines one dimension. Now, the space can be divided into two parts by using a hyperplane, and for a binary classification task the samples can be classified based on which side of the hyperplane they are [22].

The hyperplane to separate the two classes is chosen by maximizing the margin between the hyperplane and the closest data points on either side of the plane [7]. However, in real-world problems the data is often not linearly separable, which means that the data points cannot be perfectly separated into the two classes using a hyperplane in the feature space [13]. In this case, the hyperplane classifier also attempts to minimize a cost from incorrectly classified data points [7].

In addition, support vector machines can model more complex, nonlinear classification tasks. This is achieved by transforming the feature space into another, higher dimensional space, where hyperplanes for the classification translate into decision boundaries that are nonlinear in the feature space [7]. Support vector machines are therefore applicable for a wide variety of tasks [32].

4.1.5 Neural network

Neural networks are a biologically inspired method for machine learning, and, as the name suggests, they can be represented as a graph of processing units called neurons [15]. While there exist many different neural network architectures for solving different machine learning tasks, we will focus on a popular class of neural networks known as multilayer perceptrons (MLP).

Multilayer perceptrons consist of sequential layers of neurons: an input layer, one or more hidden layers, and an output layer. In classification, the input layer has one neuron for each input feature of the samples to be classified, and the output layer has one neuron for each class, each indicating the probability for the input to belong to that class. [15]. On the other hand, the number of hidden layers and the number of neurons on each layer should be chosen on a case by case basis depending on the complexity of the relationships between the input features and the output classes [28]. An example of a multilayer perceptron architecture is shown in Figure 4.2.

Each neuron in a multilayer perceptron is connected to the other neurons in the adjacent layers, and the connections between neurons have weights, which dictate how the output class of a sample is determined: The network starts with the input feature values for the neurons in the input layer. Next, for each neuron, the value is passed forward to each neuron in the following layer while multiplying it with the weight of the connection between the two neurons. The value for each neuron in the next layer is then computed by taking the sum of the incoming weighted values for that neuron, and applying an activation function. [22].

The activation function allows the network to model nonlinear relationships, and for hidden layers it is typically chosen as the sigmoid function $\phi(x) = (1 + e^{-x})^{-1}$, the hyperbolic tangent $\phi(x) = \tanh(x)$, or the rectified linear unit (ReLU) $\phi(x) = max(0, x)$ [41]. Similarly, the values for the subsequent layers are computed by again computing the weighted sums and applying the activation function. Finally, when the output layer is reached, an activation function transforms the outputs to between 0 and 1 to represent

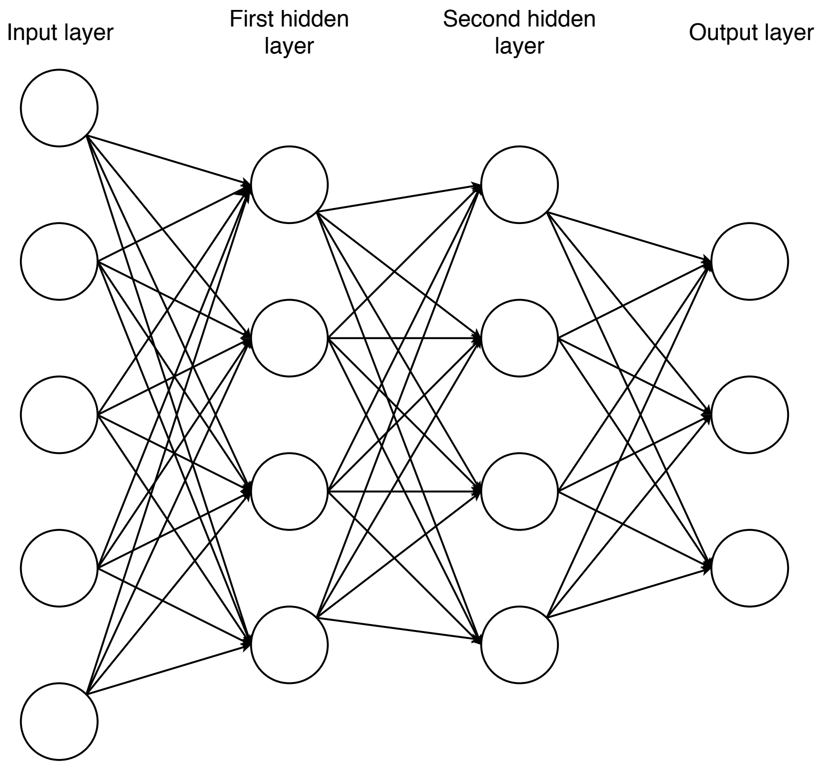

Figure 4.2: A multilayer perceptron with an input layer of 5 neurons, two hidden layers of 4 neurons each, and an output layer of 3 neurons. The neurons in each layer are connected to the neurons in the subsequent layer.

the probabilities of each class. [15].

The weights of the neural network are automatically learned by first initializing them randomly with small values. The learning algorithm then alternates between using the training data to track the classification error and adjusting the weights towards minimizing this error. This is repeated for a number of iterations or until the classification error converges. [15]. While training neural networks can be slow, the predictions of a trained network are usually fast and accurate [32].

4.2 Data preprocessing methods

Before training a machine learning model, it is often beneficial to preprocess the data, which includes applying transformations to the features, constructing new features based on the previous ones, or discarding features that are not useful. Performing appropriate data preprocessing may significantly speed up training and improve the accuracy of the machine learning model [31]. We describe two preprocessing methods, which are used in the experiments in the thesis: feature scaling, and selecting features based on their importance in decision trees.

4.2.1 Feature normalization

Feature normalization is a standard preprocessing step before using the data to train a machine learning model. If no normalization takes place, some features can have values that are much larger than the values of other features. In this case, for many machine learning methods, the features with much larger values would have significantly more impact on the output than features with smaller values. Normalizing the feature values makes all features initially contribute equally to the output, and allows the learning algorithm to decide, which features are more important. [46].

The features are often scaled by standardizing the mean and variance of each feature. Typically, the mean is transformed to zero and the features are scaled to unit variance by computing the scaled feature values as $x' = \frac{x-\mu}{\sigma}$, where μ and σ are the observed mean and standard deviation values for that feature in the training dataset. When using the machine learning model on new data, the same scaling transformations are then applied to the corresponding features in the new data. [39].

Alternatively, the values of each feature can be scaled to a set range, such as the the unit range from 0 to 1. This is achieved by computing the new values as $x' = \frac{x-min}{max-min}$, where min and max represent the observed minimum and maximum values for that feature in the training dataset [39]. However, when applying the same transformations to previously unseen data, the new scaled data values are not necessarily between 0 and 1. This is because the new data may contain feature values that are smaller than or greater than any of the values that were used to train the model, which results in scaled values that are below 0 or above 1, respectively.

Scaling can have a large effect on the resulting model when using neural networks [46] or support vector machines [23]. On the other hand, decision trees are invariant to feature scaling, which also means that random forests and gradient boosted decision trees are not affected by the scaling of the data [22].

4.2.2 Feature selection by importance in decision trees

Feature selection is another effective preprocessing method, which can reduce training time and improve model accuracy [29]. In this thesis, we perform feature selection by choosing the most relevant features according to their importance in decision trees. More precisely, features are selected as follows: First, we use the training data with all features to train a random forest model for malware detection. After training, the random forest model computes a value for each feature indicating how important that feature is for the classification task [39]. Finally, we select a fixed number of top features ranked by their importance values, and train a new classifier using only these features.

In order to evaluate the importance of a feature x, let us first consider a single decision tree. In this tree, we consider every node that uses x and measure the importance of that node using a measure called Gini impurity decrease [24]. This measure quantifies how well a decision tree node separates the training samples into the different target classes of the classification. We weight this score by how many samples reach this particular node in the decision tree. For a random forest, we then compute the feature importance values as the average importance in each decision tree [35].

4.3 Evaluation methods

Methods for evaluating the performance of classifiers are necessary for comparing different machine learning models. These methods also allow us to estimate how well our classifiers are able to detect whether a previously unseen file is a malware or not. This section describes the evaluation metrics and methods that are used in this thesis.

4.3.1 Performance metrics

When evaluating a classifier using a set of test files, we label each malware as a true positive (TP), if the classifier detects that the file is malicious, or as a false negative (FN), if the classifier incorrectly classifies the sample as a clean file. Similarly, we label each clean file as a true negative (TN), when the file is correctly detected as not malware, or as a false positive (FP), when the classifier incorrectly detects the file as a malware. The accuracy of the classifier is then defined as the proportion of correctly classified samples, $accuracy = \frac{TP+TN}{TP+FN+TN+FP}$, where TP, FN, TN, and FP are the number of true positive, false negative, true negative, and false positive samples, respectively [17].

However, in malware detection the goal is not always to classify correctly as many of the files as possible. Instead, antivirus companies usually prioritize avoiding false positives over detecting more malware [45]. For this reason, we compare our machine learning models using receiver operating characteristic (ROC) graphs, which depict the tradeoff between detections and false alarms. This tradeoff is visualized by plotting the false positive rate $FPR = \frac{FP}{TN+FP}$ on the x-axis and the true positive rate $TPR = \frac{TP}{TP+FN}$, on the y-axis [17].

A discrete classifier, which outputs whether a given file is malware or not, has fixed FPR and TPR and is thus depicted as point in the ROC graph. In contrast, we represent each classifier as a curve in the ROC graph. For this purpose, we use the machine learning models to produce scores between 0 and 1 that indicate how likely each file is to be malware. For calculating accuracy, the files are typically classified as malware if their score is more than 0.5. However, we may also use any other threshold for the scores to create another discrete classifier, and varying the threshold from 0 to 1 produces a series of classifiers, tracing a curve in the ROC graph, known as the ROC curve [17]. The ROC curve ranges from point (0,0) to point (1,1), because the classifier corresponding to threshold 0 has both a FPR of 1 and a TPR of 1, as all files are detected as malware. Similarly for the classifier corresponding to threshold 1, the FPR and TPR are both 0 as no files are detected as malware. An example of an ROC curve is shown in Figure 4.3.

In order to numerically evaluate the ROC curves of different machine learning models, we compute the area under the ROC curve (AUC). A model that correctly gives a higher score for all malicious files than any of the clean files has an AUC of 1, while a model with random scores for samples achieves on average an AUC of 0.5 [17].

4.3.2 Cross-validation

Cross-validation is a standard method for evaluating the performance of statistical models [25]. This thesis uses cross-validation, since simply computing a metric such as accuracy or AUC using only training data is not a good method for evaluating a machine learning model. This is because the model might overfit, essentially memorizing the correct answers for samples it is trained on rather than learning general predictive rules [14]. This would yield a high accuracy on the training data, but it does not indicate good performance for any other samples.

To account for overfitting, a machine learning model should instead be evaluated on data that is not used in the training of the model. In K-fold

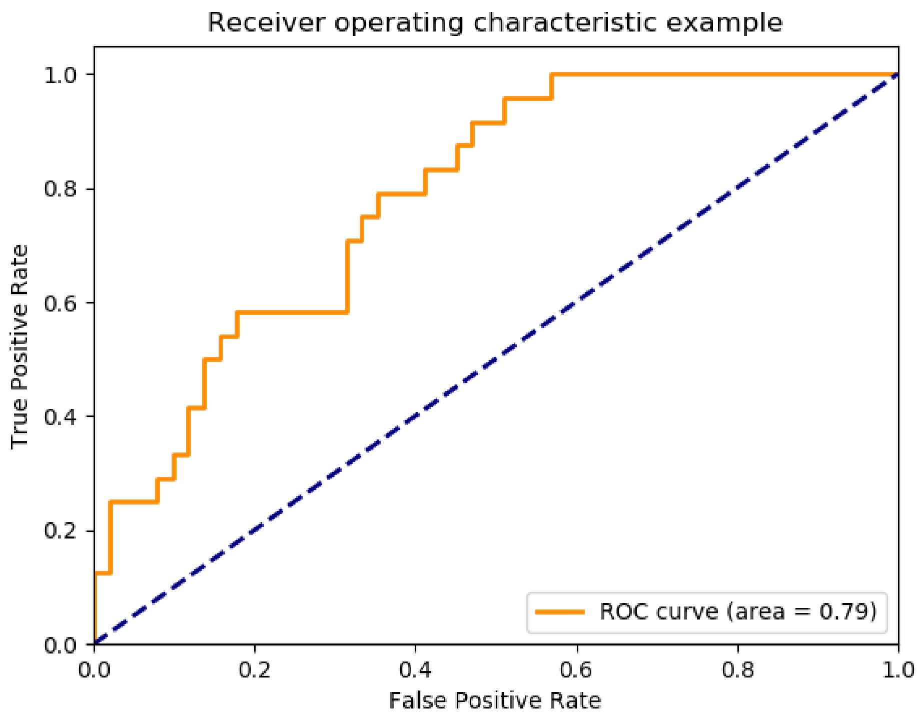

Figure 4.3: Example of an ROC curve [39]. The blue dotted line represents the performance of classifiers that only use random guessing.

cross-validation, this is achieved by partitioning the dataset into K parts of equal size. Then, $K-1$ parts are used to train a model while the last part is used to test the performance of the model. This is repeated K times, such that each part is used once to evaluate the model after it has been trained on the $K-1$ other parts. Finally, the performance results of the model across the K folds are aggregated to form a more accurate estimate of how well the model performs in general [15].

4.4 Methods for concept drift adaptation

As concept drift is prevalent in malware detection, we need methods for updating our machine learning models to deal with changes in the data over time [27]. In this section, we describe three concept drift adaptation methods that are used in this thesis: regularly retraining the model, adjusting model classification threshold, and a hybrid method combining these two. In addition, we present windowed variants of these methods, where only a number of most recent samples are considered when updating the model.

4.4.1 Model retraining

A standard solution to dealing with constantly changing data is to continuously acquire new samples and use them for training the machine learning model [19]. To apply this to the static classifiers that are used in this thesis, we regularly retrain our model using updated training data that includes the new samples.

However, retraining a model from scratch can be computationally expensive, and if only a few new training samples are included in the training set, the model will likely not improve very much. Therefore, we limit how often the model can be retrained by fixing how many new samples are needed before retraining the model: we do not retrain the model, if less than 500 new samples have been obtained since the previous time the model was trained. Furthermore, we keep track of the running accuracy of the model, and only retrain if the accuracy on the latest 1000 samples is below a set threshold.

From here on, this method is referred to as the RETRAIN method in this thesis. Pseudocode for this method is presented in Algorithm 1, which we apply using the parameters $N_{window} = 1000$ and $N_{retrain} = 500$. We discuss the different choices of accuracy threshold parameter θ later in Section 5.2.2.

As we obtain new samples with known labels, we compare the predictions with the true labels to estimate the accuracy of the current model. When the model is retrained it learns to also classify the new training samples correctly, which means that those samples can no longer be used to accurately evaluate the new model. Therefore, when computing the accuracy of the adapting model, we do not reclassify past samples using the new version of the model but we use the classifications that were obtained before retraining.

4.4.2 Model threshold adjusting

As another method for adapting to changes in the data, we try adjusting the model threshold that determines if files are classified as malware or as clean files. First of all, the machine learning models in this thesis produce classification scores between 0 and 1, where 1 indicates that the input file is malware and 0 means it is clean. The more confident the model is about the classification, the closer the score is to either 0 or to 1. By default, files that receive a score of more than 0.5 or are classified as malware, while the remaining files are classified as clean.

Algorithm 1 Procedure for evaluating the RETRAIN method.

```
Parameters:
    $D_{train}$: Training dataset
    $D_{test}$: Test dataset
    $N_{window}$: Window size for running accuracy of consecutive samples
    $N_{retrain}$: Minimum number of new samples before retraining the model
    $\theta \in [0, 1]$: Accuracy threshold for choosing when to retrain the model
1: procedure RETRAIN($N_{window}, N_{retrain}, \theta$)
2:     $N \leftarrow N_{window} - 1$
3:     $previous\_retrain \leftarrow 0$
4:     Classify samples $D_{test}[1, \dots, N]$
5:     for $i = 1, \dots, |D_{test}| - N$ do
6:         Classify $D_{test}[i + N]$
7:         $accuracy \leftarrow$ accuracy of classifications of $D_{test}[i, \dots, i + N]$
8:         if $i - previous\_retrain \geq N_{retrain}$ and $accuracy \leq \theta$ then
9:             Retrain the model using $D_{train}$ and $D_{test}[1, \dots, i]$
10:            $previous\_retrain \leftarrow i$
```

However, using the threshold of 0.5 is not always optimal, and we may instead choose any other threshold between 0 and 1 to decide how the files are classified [17]. As the data changes over time, the best choice for model threshold may also keep changing. We deal with this by using a method that continuously adjusts the threshold of our model for malware detection. We will refer to this as the THRESHOLD method, and Algorithm 2 describes it in pseudocode. This is computationally less expensive than RETRAIN, since with THRESHOLD the machine learning model is only trained once using the original training data.

Whenever we receive new samples that are known to be malware or clean, THRESHOLD selects a new threshold for our model using both the new and the earlier samples. As the new threshold, we choose a threshold that maximizes the difference between the true positive rate and the false positive rate of the classifier. This maximum difference between the TPR and FPR is known as the Youden Index, and the threshold achieving it minimizes the combined cost of false negatives and false positives. We use the same cost for false negatives and false positives for simplicity, but the same method can be applied with a generalized Youden Index to choose a higher relative cost for false positives, for instance [49].

Algorithm 2 Procedure for evaluating the THRESHOLD method.

```
Parameters:
  $D_{train}$: Training dataset
  $D_{test}$: Test dataset

 1: procedure THRESHOLD
 2:     $\sigma \leftarrow 0.5$
 3:     $S \leftarrow$ classification scores for $D_{train}$
 4:     for $i = 1, \ldots, |D_{test}|$ do
 5:         $score \leftarrow$ classification score for $D_{test}[i]$
 6:         if $score > \sigma$ then
 7:             Classify sample $i$ as malware
 8:         else
 9:             Classify sample $i$ as clean
10:         $T \leftarrow$ classification scores for $D_{test}[1, \ldots, i]$
11:         $\sigma \leftarrow$ threshold maximizing $TPR - FPR$ for $S$ and $T$
```

4.4.3 Hybrid method

For a third method for concept drift adaptation, we combine the RETRAIN and THRESHOLD methods and call it the HYBRID method. With HYBRID, the model threshold is continuously updated similar to the THRESHOLD method. In addition, like with the RETRAIN method, the machine learning model is retrained when the accuracy for the last 1000 samples falls below a set threshold and at least 500 new samples have been acquired since the model was last trained. The pseudocode for this method is presented in Algorithm 3.

We use all of the available past data when retraining the model and when adjusting the model threshold. Similar to RETRAIN, when computing the running accuracy we use the model classifications that were obtained before retraining instead of using the retrained model to reclassify previous samples. However, with HYBRID we also compute the new classification scores for the previous samples using the new retrained model in order to keep updating the classification threshold of the new model.

4.4.4 Windowed variants

While our dataset is relatively small and manageable, in practice malware detection software may encounter hundreds of thousands of new malicious samples [2] and countless clean files every day. In such a setting, storing and applying machine learning on all files becomes impractical. Even if not all

Algorithm 3 Procedure for evaluating the HYBRID method.

```
Parameters:
    $D_{train}$: Training dataset
    $D_{test}$: Test dataset
    $N_{window}$: Window size for running accuracy of consecutive samples
    $N_{retrain}$: Minimum number of new samples before retraining the model
    $\theta \in [0, 1]$: Accuracy threshold for choosing when to retrain the model
 1: procedure HYBRID($N_{window}, N_{retrain}, \theta$)
 2:     $N \leftarrow N_{window} - 1$
 3:     $\sigma \leftarrow 0.5$
 4:     $previous\_retrain \leftarrow 0$
 5:     $S \leftarrow$ classification scores for $D_{train}$
 6:     Compute classification scores for $D_{test}[1, \dots, N]$
 7:     for $i = 1, \dots, |D_{test}| - N$ do
 8:         $score \leftarrow$ classification score for $D_{test}[i + N]$
 9:         if $score > threshold$ then
10:             Classify sample $i + N$ as malware
11:         else
12:             Classify sample $i + N$ as clean
13:         $accuracy \leftarrow$ accuracy of classifications of $D_{test}[i, \dots, i + N]$
14:         if $i - previous\_retrain \geq N_{retrain}$ and $accuracy \leq \theta$ then
15:             Retrain the model using $D_{train}$ and $D_{test}[1, \dots, i]$
16:             $previous\_retrain \leftarrow i$
17:             $S \leftarrow$ new classification scores for $D_{train}$
18:             Compute new classification scores for $D_{test}[1, \dots, i + N]$
19:         $T \leftarrow$ classification scores for $D_{test}[1, \dots, i]$
20:         $\sigma \leftarrow$ threshold maximizing $TPR - FPR$ for $S$ and $T$
```

of the files are saved, the cumulative amount of data over time eventually becomes too large for frequently retraining machine learning models on.

Instead, we will experiment with windowed methods, where the machine learning model is updated using only a number of most recent samples. This allows us to discard older files and speed up the process by decreasing the amount of data used for adjusting the model. On the one hand, windowed methods are less stable, as less data is used for training the model. On the other hand, windowed methods have the potential advantage of adjusting faster to changes in the data, because the model forgets about older and possibly no longer applicable information about the data. [19].

For the three methods of adapting to concept drift in this thesis, we create analogous windowed variants called WINDOWED RETRAIN, WINDOWED THRESHOLD, and WINDOWED HYBRID. With WINDOWED RETRAIN, we choose when to retrain the model as with RETRAIN, but instead of using all of the previous data to retrain the model, we only use 10000 most recent files. Similarly, WINDOWED THRESHOLD operates like THRESHOLD and the model is also trained on the first half of our dataset, but the windowed variant only looks at the classification scores of the latest 10000 samples when adjusting the threshold of the model. As before, the WINDOWED HYBRID method combines the two other methods by only using the previous 10000 files whenever retraining the model and by only using the most recent 10000 files to adjust the model threshold.

Chapter 5

Experimental evaluation

This chapter presents the experimental results in our thesis. First, we use cross-validation to compare different machine learning methods for malware detection. Then, we analyze the effect of using temporally ordered data instead of cross-validation. Finally, we compare methods for adjusting the classifier to concept drifts in the data.

5.1 Comparison of cross-validated classifiers

In our first experiments, we compare four different machine learning classifiers for malware detection: random forest, gradient boosted decision trees, neural network, and support vector machine. We evaluated the performance of each classifier by using 5-fold cross-validation, which means that for each fold, we use 80% of the files in our dataset for training and the remaining 20% for evaluating the models. First, we describe the implementation details and what model parameters were chosen. Then, we present the results of our experiments.

5.1.1 Experiment setup

For each split of the data into training and test sets, we extracted features that were present in the training set, as described in Section 3.2. Additionally, we normalized the features so that each feature had a mean value of 0 and a variance of 1. Once the normalized features were created, we used the data to train and compare classifiers. All models were trained using the same machine with an Intel Core i7-7600U processor to ensure that the training time durations are comparable between different models.

We used scikit-learn[1] machine learning library for Python programming language to create our random forest, neural network, and support vector machine models for malware detection. On the other hand, for gradient boosted decision trees we decided to use a gradient tree boosting library called XGBoost, as it has been used in many winning solutions in machine learning competitions [10].

We chose the following hyperparameters for each model based on pilot experiments: For random forest, we used 300 fully grown decision trees, while for gradient boosted decision trees we used 1000 decision trees with a maximum depth of 6. Both random forest and gradient boosted decision trees were trained using parallel computing with 4 threads running simultaneously. For neural network, we used a multilayer perceptron with one hidden layer consisting of 200 neurons, a regularization parameter $\alpha = 0.001$, and a maximum number of 500 iterations. Finally, for support vector machine, we had probability estimates enabled, because only having predictions of files being either malware or benign is not enough but the probabilities are needed for ROC curves and for later experiments on adjusting the model to concept drift. For all other hyperparameters that are not mentioned here, we used default values in the library implementations of the models.

5.1.2 Classifier comparison

The results between different models are somewhat similar, as the accuracy of each model in each fold of the cross-validation was between 97% and 99%. The average results for each model are presented in Table 5.1. Gradient boosted decision trees achieved the best average accuracy and AUC among the models in our experiment with an average accuracy of 98.8% after completing training in two hours.

On the other hand, random forest was the fastest model to train as it took only around one minute to train, and it achieved an average accuracy of 98.0%. In order to speed up our model, we decided to reduce the number of features by selecting 1000 features, which our random forest model deemed most important. In our pilot experiments, this dimensionality reduction by feature importance in random forest performed better than using principal component analysis to construct new features that contain most of the information in the data, or choosing features that are best individual indicators of whether a file is malware or not.

We performed another cross-validation comparison as before, but this time we trained first a random forest model and then trained another ma-

[1]`https://scikit-learn.org/stable/`

Table 5.1: Average accuracy, area under ROC curve (AUC), and training time for different machine learning classifiers when using 5-fold cross-validation. The compared classifiers are random forest (RF), gradient boosted decision trees (XGBoost), multilayer perceptron (MLP), and support vector machine (SVM). The best accuracy and AUC are achieved by XGBoost, whereas random forest is the fastest to train.

Classifier	**Accuracy**	**AUC**	**Training time**
RF	0.980	0.9981	1 min
XGBoost	0.988	0.9990	1 h 59 min
MLP	0.978	0.9966	20 min
SVM	0.974	0.9960	8 h 3 min

Table 5.2: Average accuracy, AUC, and training time for 5-fold cross-validation, when first using random forest to select 1000 most important features and then training another classifier using these features.

Classifier	**Accuracy**	**AUC**	**Training time**
RF+RF	0.981	0.9981	2 min
RF+XGBoost	0.987	0.9989	21 min
RF+MLP	0.979	0.9965	5 min
RF+SVM	0.976	0.9962	55 min

chine learning model, which only used the 1000 most important features chosen by the random forest. The results are presented in Table 5.2. The average accuracy and AUC of each model are very close to the respective values in Table 5.1, where features were not first selected by a random forest. However, the total training times for gradient boosted decision trees, multilayer perceptron, and support vector machine were considerably shorter when first reducing the number of features using a random forest.

On the other hand, instead of focusing on accuracy, avoiding false positives is usually considered more important in malware detection, as frequent false alarms significantly reduce the usability of an anti-malware software [1]. However, the magnified average ROC curves for the different classifiers in Figure 5.1 show that gradient boosted decision trees are the best model for malware detection regardless of what false positive rate is chosen. For instance, at a fixed false positive rate of 0.5%, this model is able to detect 96.2% of the previously unseen malware in the test set, while the other models' detection rates are 3-7 percentage points lower. Therefore, in subsequent experiments in this thesis, we will focus solely on the gradient boosted decision trees model, which uses features selected by a random forest to reduce the training time of the model.

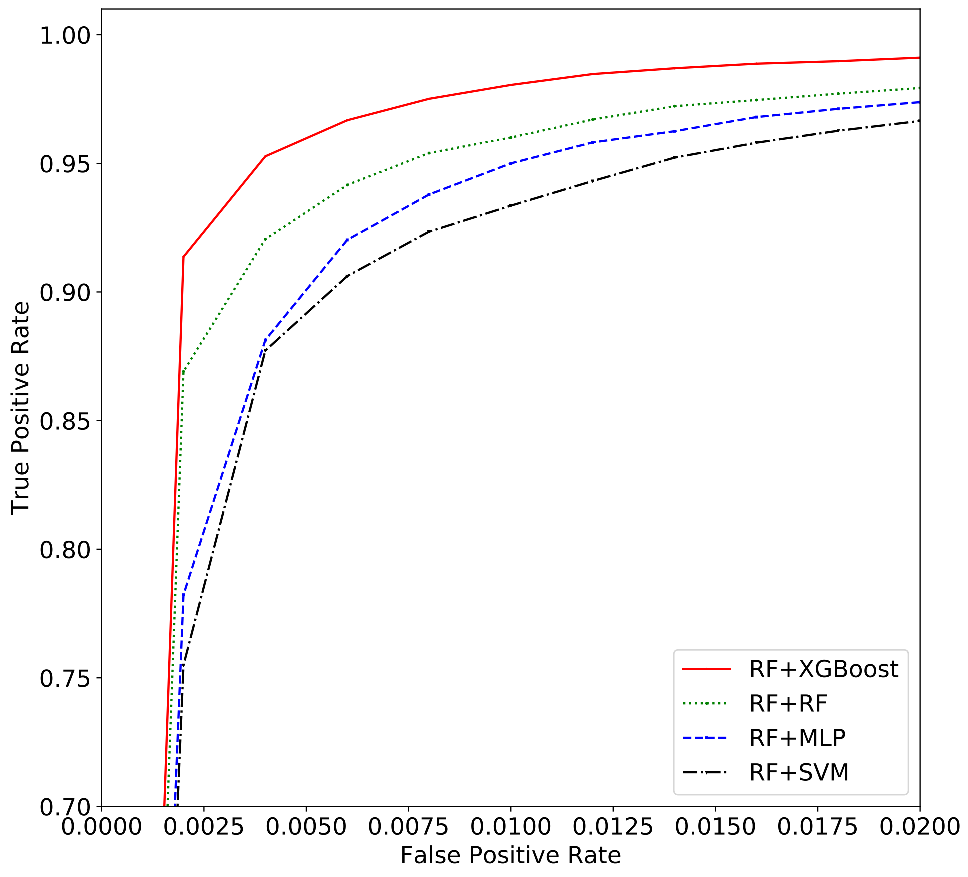

Figure 5.1: Magnified average ROC curves for the different classifiers in our cross-validation experiment. For any false positive rate, gradient boosted decision trees have a higher true positive rate than the other classifiers.

5.2 Comparison of concept drift adaptation methods

This section presents the results of experiments using the concept drift adaptation methods described in Section 4.4. First, we show that concept drift is present in our task of malware detection. Then, we discuss how choosing a threshold for when to train the model affects the performance. Finally, we compare the different methods for concept drift adaptation, as well as their variants that only consider most recent files by using a sliding window.

5.2.1 Effect of temporally ordering data

A problem with using cross-validation for estimating the performance of malware detector is that with this approach the model can use information about present and future files when classifying a sample. In a real scenario, the task is more difficult as the machine learning model has to learn from past files how to detect if files created in the future are malware or not. Indeed, prior works on malware detection have shown that cross-validation tends to give considerably higher model accuracy estimates than temporally ordered splits into training and testing data sets [27, 37]. However, a study by Bach and Maloof [3] did not find evidence of concept drift in a small-scale experiment on malware detection using static analysis of files.

To verify the effect of concept drift on our static features and dataset, we tested the gradient boosted decision trees model from our cross-validation experiments on temporally ordered data. We sorted the files by when they were first seen by F-Secure, and used the first half of files for training and the other half for testing. For later analysis in the thesis, we refer to this as the BASELINE method. This model achieved an average accuracy of 0.967, and an AUC of 0.9966. In comparison, when the same model was evaluated using 2-fold cross-validation that also uses half of the data for training and the other half for testing, the average accuracy was 0.984 and the average AUC was 0.9985. Thus, ordering the data temporally impairs the performance of the model, which indicates concept drift in our dataset of malware and benign files.

5.2.2 Choice of retraining threshold

As described in Section 4.4, our RETRAIN and HYBRID methods retrain the machine learning model when the accuracy for recent samples falls below a set threshold. If the retrain threshold is set too low, the model is never retrained, which means that there is no improvement in accuracy compared to the BASELINE method. On the other hand, if the threshold is set too high, the model will be retrained very frequently, which is computationally expensive. Thus, we need a threshold with a balance between method accuracy and how often the model is retrained.

To find a good threshold, we tested RETRAIN method with thresholds 0.89, 0.90, 0.91, ..., 1.0, and computed the method accuracy as well as how often the model was retrained for each threshold. The results of this experiment are shown in Figure 5.2. Choosing a higher threshold means that the model has to be retained more often, but also increases the accuracy. However, increasing the threshold above 0.98 does not significantly improve the accuracy but does increase the number of times the model is retrained. Therefore, we select the threshold of 0.98 for our RETRAIN and HYBRID methods.

5.2.3 Method comparison results

We trained each of the methods BASELINE, THRESHOLD, RETRAIN, and HYBRID on the first half of our temporally ordered set of files and evaluated using the second half. To analyze how the accuracy of models changed over time, we kept track of the accuracy for 1000 consecutive files over time. Figure 5.3 shows the running accuracy for each method as well as when the machine learning model was retrained with RETRAIN, and HYBRID methods. While BASELINE has a few significant drops in accuracy, these drops are less pronounced for the methods adapting to concept drift.

All three adaptation methods achieved better overall accuracy than BASELINE. We present the total accuracy on the test set for each method in Table 5.3. While the THRESHOLD method does not require the model to be retrained, RETRAIN and HYBRID methods had higher accuracy most of the time. RETRAIN and HYBRID methods achieved very similar accuracy, but on average the accuracy was slightly higher for the HYBRID method. In addition, with RETRAIN the model was retrained 23 times whereas HYBRID retrained the model 19 times.

5.2.4 Comparison with windowed methods

We performed the same experiment with the windowed methods WINDOWED THRESHOLD, WINDOWED RETRAIN, and WINDOWED HYBRID. Figure 5.4 shows the running accuracy on 1000 consecutive files for BASELINE and the three windowed methods. The markers in the figure show when WINDOWED RETRAIN and WINDOWED HYBRID methods retrained the machine learning

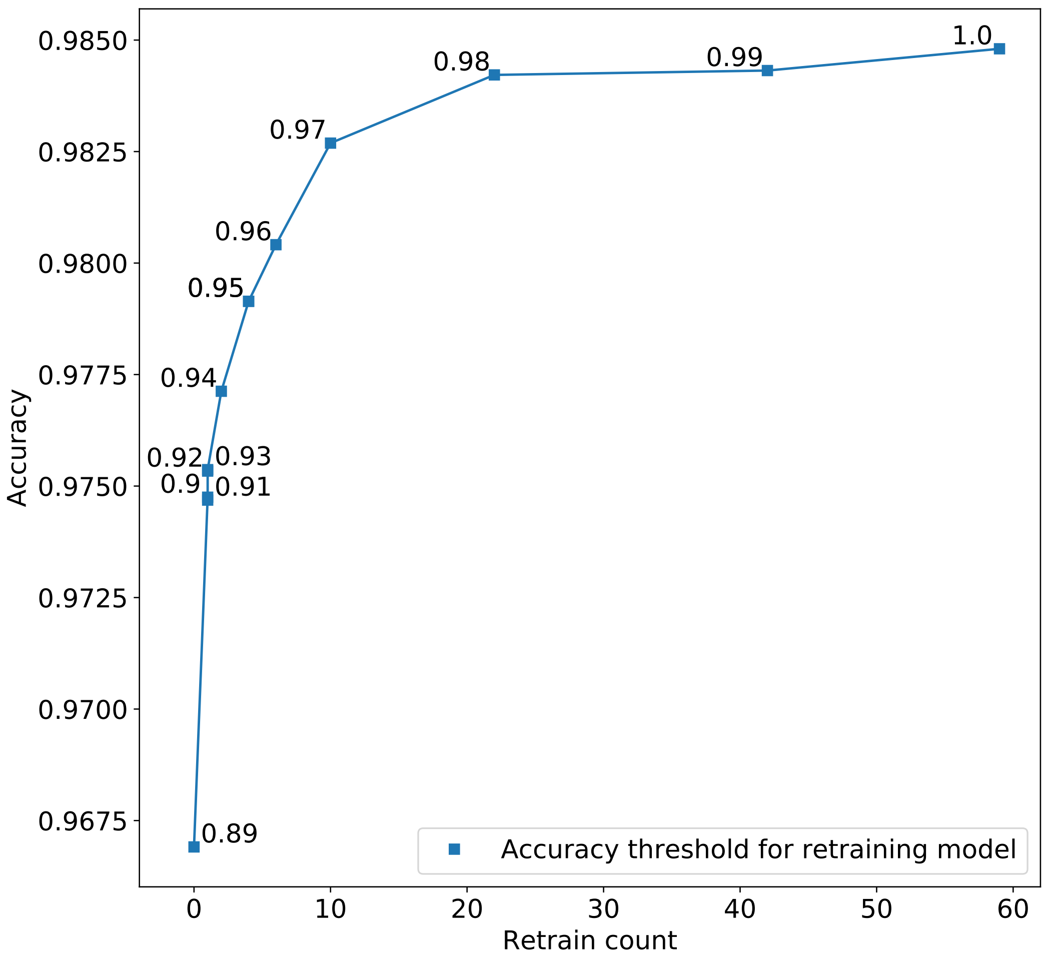

Figure 5.2: Accuracy and retrain count for different accuracy thresholds that decide when to retrain the model. The model is retrained whenever the running accuracy for 1000 consecutive samples falls below the threshold, and there are at least 500 new samples since the model was last retrained. Choosing a higher threshold leads to a higher accuracy but also increases the number of times the model has to be retrained. A threshold of 0.89 or lower means that the model is not retrained, as the running accuracy never falls below 0.89. On the other hand, a threshold of 1.0 means that the model is retrained every 500 samples regardless of its performance. Choosing a threshold higher than 0.98 notably increases the retrain count, while the accuracy does not improve significantly.

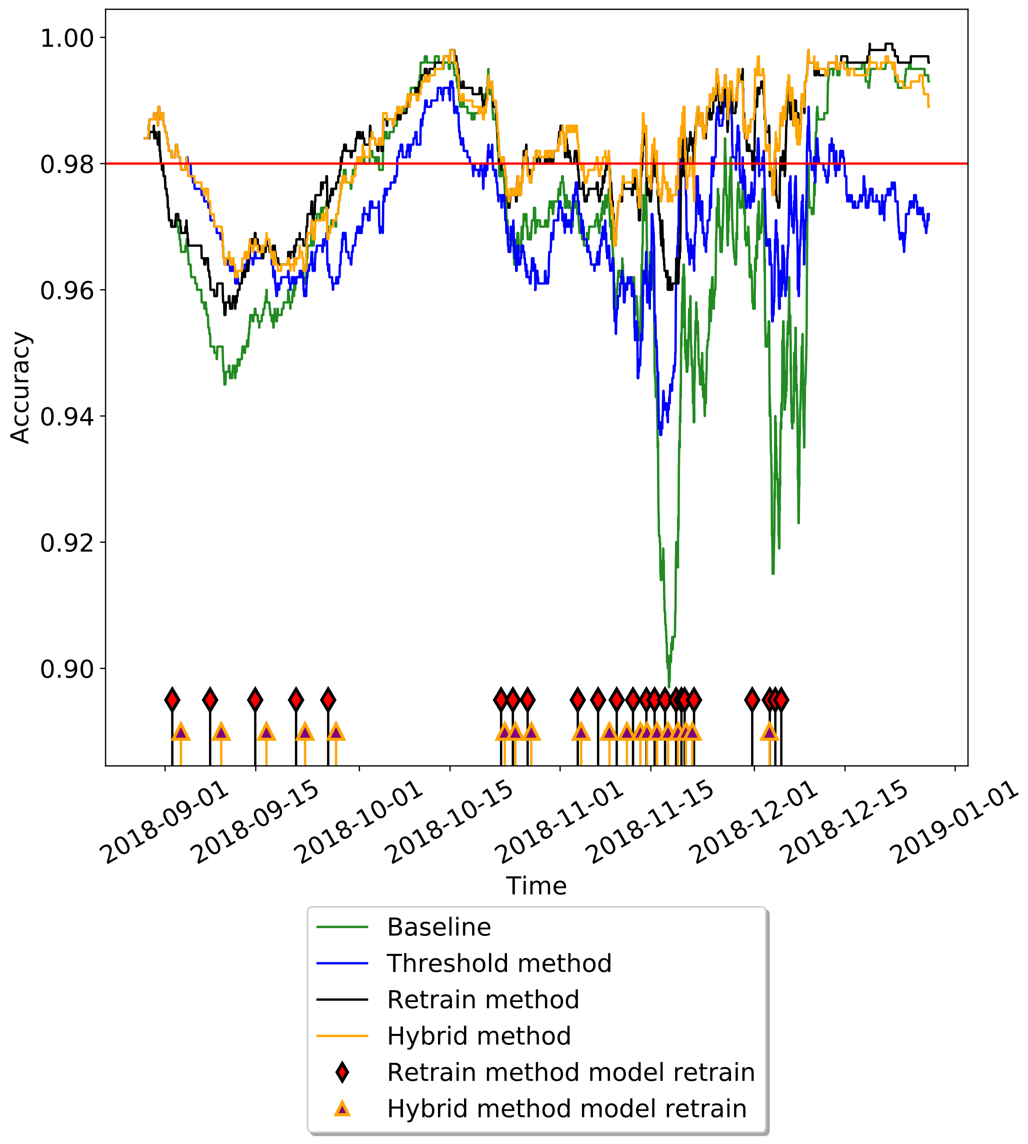

Figure 5.3: Running accuracy of 1000 consecutive samples for BASELINE, THRESHOLD, RETRAIN, and HYBRID methods. The machine learning model was retrained with RETRAIN and HYBRID methods every 500 samples if the accuracy was below 0.98. The markers at the bottom indicate the times when the model was retrained. The BASELINE method has large spikes indicating sudden decreases in accuracy for files seen in the middle of November and in early December. In comparison, THRESHOLD, RETRAIN, and HYBRID methods are able to adapt and maintain higher accuracy. Most of the time, HYBRID and RETRAIN methods achieve the best accuracy.

Table 5.3: Total accuracy for the different methods of adapting to concept drift. For each method, the model was initially trained using the first half of the data and the accuracy was evaluated using the second half.

Method	**Accuracy**
BASELINE	0.967
THRESHOLD	0.973
RETRAIN	0.984
HYBRID	0.985

Table 5.4: Total accuracy for the windowed versions of methods for adapting to concept drift. The accuracy for each method was evaluated using the second half of files in our dataset.

Method	**Accuracy**
WINDOWED THRESHOLD	0.973
WINDOWED RETRAIN	0.981
WINDOWED HYBRID	0.981

model. Compared to the non-windowed methods in Figure 5.3, the windowed methods have slightly lower accuracy most of the time.

In some periods, the BASELINE method performs better than WINDOWED RETRAIN and WINDOWED HYBRID. This could be explained by the fact that the model in BASELINE was trained using half of the dataset for 30733 files, whereas the models in WINDOWED RETRAIN and WINDOWED HYBRID methods were trained and retrained using only 10000 files. However, like the non-windowed methods for concept drift adaptation, the windowed variants are also able to mitigate the sudden drops in accuracy that appear with BASELINE.

Overall, the windowed methods are also an improvement to the BASELINE method by achieving higher accuracy. Table 5.4 gives the total accuracy for each of the windowed methods. While the windowed versions are more scalable, the RETRAIN and HYBRID methods in Table 5.3 achieved better accuracy in our experiment. This implies that the windowed variants would benefit from more closely mimicking the behaviour of the non-windowed methods by using a larger window size.

While WINDOWED RETRAIN and WINDOWED HYBRID achieved similar accuracy, WINDOWED HYBRID retrained the machine learning model slightly fewer times. The model retrain counts for RETRAIN, HYBRID, WINDOWED

Table 5.5: The number of times different methods retrained the machine learning model when evaluated on the second half of our dataset. While the windowed variants retrained the model more often, each time the training took approximately the same amount of time. In contrast, with the non-windowed versions, training took increasingly longer amounts of time each time the model was retrained because all of the accumulated previous data was used for training. The HYBRID and WINDOWED HYBRID did not need to retrain the model as often as the respective RETRAIN methods.

Method	**Retrain count**
RETRAIN	23
HYBRID	19
WINDOWED RETRAIN	32
WINDOWED HYBRID	30

RETRAIN, and WINDOWED HYBRID methods are presented in Table 5.5. Overall, the HYBRID methods outperform the respective RETRAIN methods by having similar or better accuracy while retraining the model fewer times. While the windowed variants retrained the model more often, each retraining takes less time than for the non-windowed methods. Instead, the non-windowed versions train the model using increasing amounts of data as more files are collected over time.

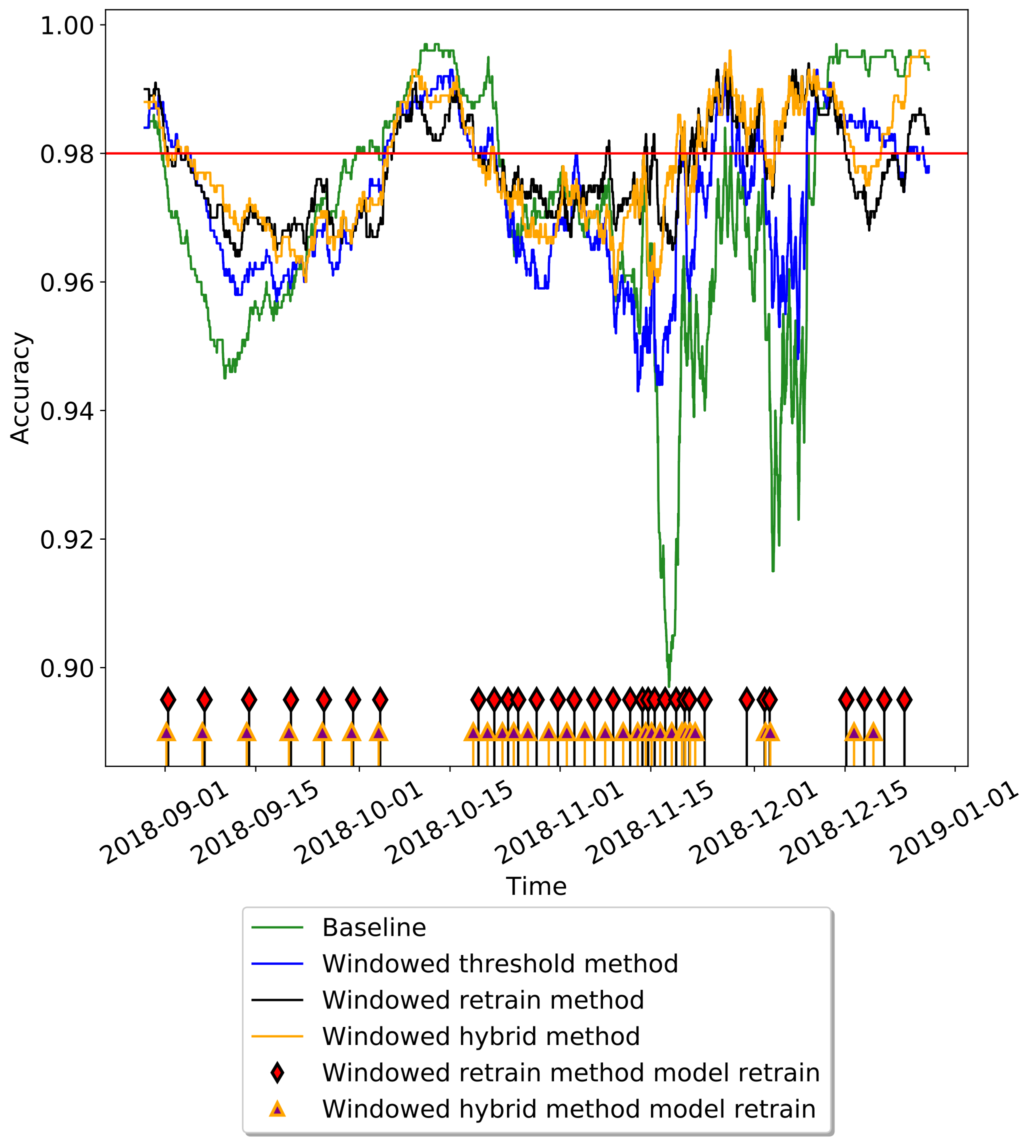

Figure 5.4: Running accuracy of 1000 consecutive samples for BASELINE, WINDOWED THRESHOLD, WINDOWED RETRAIN, and WINDOWED HYBRID methods. The markers at the bottom indicate the times when the model was retrained with WINDOWED RETRAIN and WINDOWED HYBRID methods.

Chapter 6

Conclusions and discussion

In this Master's thesis, we created a classifier for malware detection. For this purpose, we collected a dataset of 19648 malware and 41818 benign files and extracted features using static analysis of the files. In our experiments, we compared four different classifiers: random forest, gradient boosted decision trees, neural network, and support vector machines. Gradient boosted decision trees achieved higher accuracy and AUC than the other methods. Furthermore, by using a random forest model to select most important features before training the gradient boosted decision trees model, we were able to reduce the total training time of our classifier. These results support previous research showing that gradient boosted decision trees perform well in the task of malware detection using static analysis [30].

Additionally, we tested the classifier on files that were sorted by when they were first seen by F-Secure. The classification accuracy was lower on temporally ordered data than when using cross-validation, which indicates that concept drift should be taken into account in malware detection. To deal with the concept drift, we then compared RETRAIN, THRESHOLD, and HYBRID methods that adapt the classifier to changes in the data over time. All three methods improved the accuracy of the classifier, and the best results were achieved by the HYBRID method, which merges the two other methods. These results show that combining different methods for adapting to concept drift can give a better malware detector than the individual methods.

We also compared the methods for adapting to concept drift with windowed variants that only consider most recent files. While the windowed variants are more practical when the number of files is large, in our experiments the windowed methods achieved lower accuracy than the original methods. To better assess how forgetting older files influences malware detection, more studies on windowed methods should be carried out using a larger dataset.

In this thesis, the methods for adapting to changes in the data require that we obtain labels indicating if new files are malicious or clean. However, having human analysts create a large number of these labels is time-consuming and expensive. Future work could study how labeling only a fraction of the new files would impact the classifier over time and how to choose which files should be labeled by an analyst. In addition, it remains to be studied whether the classifier becomes unreliable if it is updated by using labels that are automatically created. Alternatively, further research could incorporate dynamic analysis of files for potentially more accurate malware detection.

Bibliography

[1] AV-Test. Test modules under windows, 2019. https://www.av-test.org/en/about-the-institute/test-procedures/. Accessed 20.9.2019.

[2] AV-Test. Malware statistics & trends report, 2019. https://www.av-test.org/en/statistics/malware/. Accessed 26.7.2019.

[3] Stephen H Bach and Marcus A Maloof. Paired learners for concept drift. In *2008 Eighth IEEE International Conference on Data Mining*, pages 23–32. IEEE, 2008.

[4] Zahra Bazrafshan, Hashem Hashemi, Seyed Mehdi Hazrati Fard, and Ali Hamzeh. A survey on heuristic malware detection techniques. In *The 5th Conference on Information and Knowledge Technology*, pages 113–120. IEEE, 2013.

[5] BBC News. Where has ransomware hit hardest?, 2017. https://www.bbc.com/news/world-39919249. Accessed 5.8.2019.

[6] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[7] Christopher JC Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.

[8] Carnegie Mellon University. CERT Advisory CA-1999-02 Trojan Horses, 1999. https://resources.sei.cmu.edu/asset_files/WhitePaper/1999_019_001_496184.pdf. Accessed 10.9.2019.

[9] Ero Carrera. pefile, 2018. https://github.com/erocarrera/pefile. Accessed 18.9.2019.

[10] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[11] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, and Yuan Tang. Xgboost: extreme gradient boosting. *R package version 0.4-2*, pages 1–4, 2015.

[12] Fred Cohen. Computer viruses: theory and experiments. *Computers & security*, 6(1):22–35, 1987.

[13] Nello Cristianini, John Shawe-Taylor, et al. *An introduction to support vector machines and other kernel-based learning methods.* Cambridge university press, 2000.

[14] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM computing surveys*, 27(3):326–327, 1995.

[15] Ke-Lin Du and Madisetti NS Swamy. *Neural networks and statistical learning.* Springer Science & Business Media, 2013.

[16] Eastlake, Donald and Jones, Paul. US secure hash algorithm 1 (SHA1), 2001.

[17] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[18] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[19] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):44, 2014.

[20] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 5(02):56, 2014.

[21] Thomas Hancock, Tao Jiang, Ming Li, and John Tromp. Lower bounds on learning decision lists and trees. *Information and Computation*, 126 (2):114–122, 1996.

[22] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction.* Springer, 2nd edition, 2009.

[23] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification, 2003.

[24] Hemant Ishwaran. The effect of splitting on random forests. *Machine Learning*, 99(1):75–118, 2015.

[25] Peter H. M. Janssen, Petre Stoica, T Söderström, and Pieter Eykhoff. Model structure selection for multivariable systems by cross-validation methods. *International Journal of Control*, 47(6):1737–1758, 1988.

[26] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 625–642, 2017.

[27] Alex Kantchelian, Sadia Afroz, Ling Huang, Aylin Caliskan Islam, Brad Miller, Michael Carl Tschantz, Rachel Greenstadt, Anthony D Joseph, and JD Tygar. Approaches to adversarial drift. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 99–110. ACM, 2013.

[28] Saurabh Karsoliya. Approximating number of hidden layer neurons in multiple hidden layer BPNN architecture. *International Journal of Engineering Trends and Technology*, 3(6):714–717, 2012.

[29] Ron Kohavi and George H John. Wrappers for feature subset selection. *Artificial intelligence*, 97(1-2):273–324, 1997.

[30] J Zico Kolter and Marcus A Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7(Dec):2721–2744, 2006.

[31] SB Kotsiantis, Dimitris Kanellopoulos, and PE Pintelas. Data preprocessing for supervised leaning. *International Journal of Computer Science*, 1(2):111–117, 2006.

[32] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24, 2007.

[33] Simon Kramer and Julian C Bradfield. A general definition of malware. *Journal in computer virology*, 6(2):105–114, 2010.

[34] Cheng Li. A Gentle Introduction to Gradient Boosting, 2019. `http://www.chengli.io/tutorials/gradient_boosting.pdf`. Accessed 9.10.2019.

[35] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. Understanding variable importances in forests of randomized trees. In *Advances in neural information processing systems*, pages 431–439, 2013.

[36] Mead, Steve. Unique file identification in the national software reference library. *Digital Investigation*, 3(3):138–150, 2006.

[37] Brad Miller, Alex Kantchelian, S Afroz, R Bachwani, R Faizullabhoy, L Huang, V Shankar, MC Tschantz, Tony Wu, George Yiu, et al. Back to the future: Malware detection with temporally consistent labels. *Under submission*, 2015.

[38] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[40] Matt Pietrek. Peering inside the pe: a tour of the win32 (r) portable executable file format. *Microsoft Systems Journal-US Edition*, 9(3):15–38, 1994.

[41] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[42] ReversingLabs. TitaniumScale Enterprise Scale File Analysis, 2019. `https://cdn2.hubspot.net/hubfs/3375217/ReversingLabs%20Data%20Sheets/2019-Jan-RL-TitaniumScale-Integration-Datasheet-EN.pdf`. Accessed 10.9.2019.

[43] Lior Rokach and Oded Maimon. *Data Mining with Decision Trees: Theory and Applications*. Singapore: World Scientific, 2008.

[44] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[45] Igor Santos, Yoseba K Penya, Jaime Devesa, and Pablo Garcia Bringas. N-grams-based file signatures for malware detection. *ICEIS (2)*, 9:317–320, 2009.

[46] Warren S. Sarle. Neural Network FAQ, 2002. `http://www.faqs.org/faqs/ai-faq/neural-nets/`. Accessed 23.10.2019.

[47] Michael Satran, Mark Leblanc, Colin Robertson, Karl Bridge, John Kennedy, Drew Batchelor, and Christopher Warrington. PE Format - Windows applications, 2019. `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format`. Accessed 17.9.2019.

[48] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–20. IEEE, 2015.

[49] Enrique F Schisterman, David Faraggi, Benjamin Reiser, and Jessica Hu. Youden index and the optimal threshold for markers with mass at zero. *Statistics in medicine*, 27(2):297–315, 2008.

[50] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 38–49. IEEE, 2000.

[51] Anshuman Singh, Andrew Walenstein, and Arun Lakhotia. Tracking concept drift in malware families. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, pages 81–92. ACM, 2012.

[52] KP Soman, Shyam Diwakar, and V Ajay. *Data mining: theory and practice [with CD]*. PHI Learning Pvt. Ltd., 2006.