FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Interpretability of machine learning-based results of malware detection using a set of rules
Student:	Jan Dolejš
Supervisor:	Mgr. Martin Jureček
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2021/2022

Instructions

Nowadays, machine learning algorithms are common techniques used to detect malware. However, ML algorithms are not directly incorporated into antivirus programs installed on users' systems. Therefore, it is convenient to interpret the ML results (obtained in the cloud) as a set of detection rules (e.g., as formulas: (x_1 o_1 h_1) AND ... AND (x_n o_n h_n), where x_i are features, o_i relational operators, and h_i values) to avoid the need to maintain large datasets.

Instructions:
1) Study and describe methods for creating malware detection rules.

2) Collect benign and download malware samples (in the PE file format) or use an existing dataset provided by the supervisor.
3) Use ML libraries (e.g., Scikit-learn) and apply at least three state-of-the-art ML algorithms to the dataset from step 2).
4) Implement at least two methods for creating malware detection rules to describe the results from step 3).

5) Evaluate and discuss the experimental results.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 25 January 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Interpretability of Machine Learning-based Results of Malware Detection using a Set of Rules

Jan Dolejš

Department of Information Security
Supervisor: Mgr. Martin Jureček

May 13, 2021

Acknowledgements

I would like to express my profound gratitude to my supervisor Mgr. Martin Jureček. He was a great source of knowledge as well as motivation throughout the work. Thank you.

Further thanks go to my family and friends for their continuous (and differentiable) support in all these years. Life would be tough to imagine without you.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 13, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Jan Dolejš. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Dolejš, Jan. *Interpretability of Machine Learning-based Results of Malware Detection using a Set of Rules*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstract

Machine learning methods have been quite successful in a variety of applications. Antivirus companies use them for quick and reliable malware detection, providing their users with a safer environment from ceaseless daily threats. However, machine learning methods such as deep neural networks are often considered black boxes as the reasoning behind their decisions may often be unclear. Their interpretability is important and helps understand potential errorful decisions. This thesis studies rule-learning algorithms and explores their potential to interpret the outcomes of machine learning algorithms. Two publicly available datasets with Portable Executable file attributes and tailor-made implementations of rule-learning algorithms were used throughout the work. Results showed that algorithm RIPPER is mostly successful at this task; it achieved high accuracies while maintaining compact sets of rules, making rule-learning algorithms a useful alternative to signature-based methods.

Keywords malware detection, rule-based classifiers, interpreting machine learning, PE files

Abstrakt

Metody strojového učení se prokázaly jako užitečný nástroj v řadě aplikací. Antivirové společnosti našly jejich využití i pro rychlou a spolehlivou detekci malwaru, poskytující jejich uživatelům bezpečnější prostředí před každodenními hrozbami. Metody strojového učení, jako jsou například hluboké neuronové sítě, jsou však často považovány za black boxy, jelikož důvody jejich rozhodnutí mohou být často nejasné. Jejich intepretovatelnost je důležitá a pomáhá pochopit potenciálně chybná rozhodnutí. Tato práce se zabývá algoritmy pro tvorbu pravidel a zkoumá jejich potenciál v rámci interpretace výsledků metod strojového učení. V práci bylo využito dvou veřejně dostupných datasetů, obsahujících atributy PE souborů, a na míru navržených implementací algoritmů pro tvorbu pravidel. Výsledky ukázaly, že algoritmus RIPPER je v tomto úkolu převážně úspěšný; vysokou přesnost vykazoval i při zachování kompaktních sad pravidel, což dělá z algoritmů pro tvorbu pravidel užitečnou alternativu metody založené na signaturách.

Klíčová slova detekce malwaru, klasifikátory založené na pravidlech, interpretace strojového učení, PE soubory

Contents

Introduction **1**

1 Decision Rules **3**
1.1 Formal Introduction . . . 3
1.2 Rule Characteristics . . . 5
1.3 From Trees to Rules . . . 6
1.4 Rule Learning Algorithms . . . 7
1.4.1 1R . . . 7
1.4.2 CN2 . . . 8
1.4.3 I-REP . . . 12
1.4.4 RIPPER . . . 14
1.4.5 Other Approaches . . . 18

2 Portable Executable **21**
2.1 Dos Header and Stub . . . 21
2.2 PE Header . . . 22
2.3 Optional Header . . . 22
2.4 Section Headers . . . 22

3 Implementations of Rule Based Classifiers **23**
3.1 Decision List . . . 23
3.2 1R . . . 24
3.3 I-REP . . . 24
3.4 RIPPER . . . 24
3.5 Greedy Grow . . . 25

4 Experiments **29**
4.1 Dataset Description . . . 29
4.1.1 Kozák's Dataset . . . 29

4.1.2 EMBER Dataset 30
4.2 Feature Transformation and Selection 30
4.2.1 Feature Vectorisation 30
4.2.2 Feature Hashing 31
4.2.3 Feature Selection 31
4.3 Evaluation Metrics 32
4.4 Interpreting ML results using RBCs 33
4.4.1 Pruning and Metrics 35
4.4.2 Removing Rules in RIPPER 38

5 Discussion **43**
5.1 Experiments Review 43
5.2 Comparison with Related Works 45

Conclusion **47**

Bibliography **49**

A Acronyms **55**

B Figures **57**

C Contents of enclosed SD Card **61**

List of Figures

1.1 Decision Tree - Describing a simple disjunction [13] 6

4.1 Understanding pruning metrics as 3D graphs 37

5.1 Rule coverage size – RandomForest & EMBER 44
5.2 Visualisation of rule sizes over time – RandomForest & EMBER . 44

B.1 Rule coverage size – EMBER 58
B.2 Rule size – EMBER 59

Introduction

The first-ever malicious software (malware) appeared as a school joke in 1982 [1]. Since then, malware development has become a serious business and a threat to computer and smartphone users. In 2017, ransomware called WannaCry was widely spread across the Windows operating system due to an unpatched bug [2]. Ransomware is a type of malicious software that typically demands ransom payment for returning access to a user's device [3]. Prior to this incident, many massive Distributed Denial of Service (**DDoS**) attacks have taken place. This was done using the botnet malware Mirai, which was used to enslave poorly secured IoT devices. One of the targets was Brian Krebs's website with a 620 Gbps network load. Brian Krebs is an American journalist who later discovered those responsible for the attack [4].

Two approaches exist to determine whether a file is malicious or not: static analysis and dynamic analysis [5]. Dynamic analysis requires the file to be executed, commonly in a sandbox environment, to examine its behaviour [6]. In contrast, static malware detection does not require the file to be executed and is used to examine file's attributes, e.g. opcodes, file headers, machine code, and more.

We can combine both approaches with machine learning methods for effective malware detection [7]. The advantage of this is identifying new malicious samples with high accuracy [8]. However, it is still necessary to maintain an extensive database of file hashes with either malicious or benign labels, which is ineffective and most probably unsustainable for future usage. Thus, it is practical to maintain the results as a set of detection rules.

This thesis aims to study rule-learning algorithms and measure their performance on the outcomes of machine learning methods. Machine learning methods are often considered black boxes, and we can interpret their outcomes using a set of rules. We will create our implementations of rule-learning algorithms and propose a new algorithm. The implementations will allow us to change certain parts of the algorithms and better understand their behaviours. The results will be shown using two publicly available datasets on which we

will train five machine learning models. Used datasets will contain information about Portable Executable files gathered through static analysis.

The structure of the thesis is as follows:

Chapter 1 introduces the theory necessary for rule-learning algorithms. Next, four rule-learning algorithms are described.

Chapter 2 explains the PE file format, which is later used in the datasets.

Chapter 3 describes the specifics of our implementations and introduces one additional algorithm with a different approach to rule-learning.

Chapter 4 gives details on the datasets, as well as their transformation and feature selection. Further, the experimental setup and process are described. Chapter 4 contains the evaluation of the experiments, too.

Chapter 5 further discusses the results from Chapter 4 with an additional view on the results.

Chapter 1

Decision Rules

People are used to making decisions daily – from getting up once our alarm clock starts ringing to making ourselves a bit of food once we get hungry. We do these actions quite naturally, and it is uncommon for us to think of them as rules. On the other hand, these actions could very easily be expressed as a set of *if-then* rules. They would remain as expressable as our initial thought, for example: *if* alarm clock starts ringing *then* get up or *if* hungry *then* make food.

We call these rules decision rules, and they are considered one of the more interpretable models in machine learning [9]. In [10], the authors define *interpretability* as "the ability to explain or present in understandable terms to a human". Other machine learning methods, such as deep neural networks, are often considered black boxes as the reasoning behind their decisions may often be unclear [11]. However, the interpretability of the models is very important; nowadays, they are widely used for various applications, e.g. criminal risk assessment [12] or malware detection [7]. Interpreting the results means that we can explain why a person is considered at high risk of criminal activity or why a benign file was marked malicious. This work will focus on classification rules used to classify data, as the name already suggests [13].

1.1 Formal Introduction

Above, we have only shown an elementary set of rules. The antecedent, or the precondition, is usually more complicated and is built upon many different features. The consequent, or the conclusion, gives us the ability to decide, or more importantly in this work, to classify different samples. We can generalise a single precondition in the following form.

Table 1.1: Example of the EMBER [16] dataset content

filesize	size of headers	timestamp	printables count (#)	class
16,384	512	2018-07-02	2,072	benign
40,960	4,096	2007-05-04	2,526	benign
45,568	1,536	2019-05-10	4,213	benign
156,784	1,024	2016-10-11	8,479	malware
167,936	1,024	2016-04-06	4,227	benign
245,829	1,024	2013-04-01	11,460	malware
298,486	1,024	2004-11-06	15,703	malware
773,044	512	2018-05-29	25,774	benign
817,518	1,024	1992-06-20	97,446	benign
966,594	4,096	2017-09-03	22,979	malware

Definition 1 (Condition)**.** *We define condition c as follows:*

$$c \equiv x \odot h, \tag{1.1}$$

where x is a feature, $\odot$ is a relation, and h is the searched value.

Usually, the preconditions are logically ANDed together, making it necessary for all tests to fire [9, 13, 14].

Definition 2 (Rule & Rule Size)**.** *We define rule r as follows:*

$$r \equiv c_1 \wedge \cdots \wedge c_m, \tag{1.2}$$

where m is the number of conditions for rule r. We say that rule r has a size of m.

However, the condition conjunction is not a necessity, and a single rule may be expressed by a general logical expression [13, 15].

To get a bit more familiar with the samples we will be working with, we have included Table 1.1 from the EMBER [16] dataset, which is later covered in Section 4.1.2. We could create the following rule for the samples in Table 1.1:

$$\begin{aligned} &if\ size\ of\ headers = 1024 \wedge \\ &\quad timestamp > \text{2000-01-01}\ then\ is\ malware. \end{aligned} \tag{1.3}$$

Notice, however, that this rule does not classify all samples correctly.

For a given rule, we are interested in its quality, more specifically in its coverage (support) and accuracy (confidence). We say that a rule *covers* a sample if the sample satisfies the rule's preconditions [17].

Definition 3 (Coverage)**.** *Given a set of samples S, we define the coverage of rule r as*

$$coverage(r, S) = \{s \mid s \in S, r\ covers\ s\}. \tag{1.4}$$

We may want to express this numerically.

Definition 4 (Coverage Size)**.** *Given a set of samples S, we define the coverage size of rule r as*

$$coverage\_size(r, S) = \frac{|coverage(r, S)|}{|S|}. \tag{1.5}$$

Rule (1.3) covers four samples out of ten from Table 1.1; thus, its coverage size is 40%.

For a set of samples S_i in a given class i, we are interested in the accuracy of a given rule. We understand accuracy as a metric [15], for example:

$$accuracy(r, S_i) = \frac{|coverage(r, S_i)|}{|coverage(r, S)|}, i \in \{1, \ldots, m\}, \tag{1.6}$$

where m is the number of classes in S. Rule (1.3) covers three samples from the malware class. However, it also covers one sample from the benign class. Its accuracy given by Metric (1.6) is 75%, where S_i is a set of samples from the malware class.

1.2 Rule Characteristics

Rules are said to be mutually exclusive if no two rules cover the same sample.

Definition 5 (Mutually Exclusive Rules)**.** *Given a set of samples S, we say that rule r_i and rule r_j are mutually exclusive, if*

$$coverage(r_i, S) \cap coverage(r_j, S) = \emptyset, i \neq j, i, j \in \{1, \ldots, n\}, \tag{1.7}$$

where n is the number of rules for S.

Rules are said to be exhaustive if every sample is covered by at least one rule [14].

Definition 6 (Exhaustive Rules)**.** *Given a set of samples S, we say that rules r_i are exhaustive, if*

$$\bigcup_{i=1}^{n} coverage(r_i, S) = S, i \in \{1, \ldots, n\}, \tag{1.8}$$

where n is the number of rules for S.

However, such rule restrictions are often not required, and we allow the rules to overlap and not cover the whole set. Different problems arise, some rules may contradict each other, or some of the samples may not be covered at all. Two different schemas can be used to solve this: a decision list or a decision set [9, 14].

In a decision list, the rules are ordered as follows:

$$R = [r_1, r_2, \ldots, r_n], \tag{1.9}$$

where n is the number of rules for a given list. In other words, the rules are kept in the order in which they were added. The same order is later used for classification, too [9].

A decision set does not require rules to be ordered; instead, all rules get to vote on classifying a given sample. Unfortunately, it is pretty easy to lose the interpretability with the voting schema once a decision set grows larger [9]. Thus, we will be using a decision list in this work if not stated otherwise.

1.3 From Trees to Rules

Before proceeding to the next section, let us briefly compare another popular machine learning tool – a decision tree. Decision trees are built from nodes, where each node, except the last ones, tests a feature with a given value (see Definition 1). The last node, also called a leaf, represents a decision, for example, classifying samples as benign or malicious [13]. Although the idea behind decision trees is quite simple, they may turn out to be quite complex and hard to interpret [18].

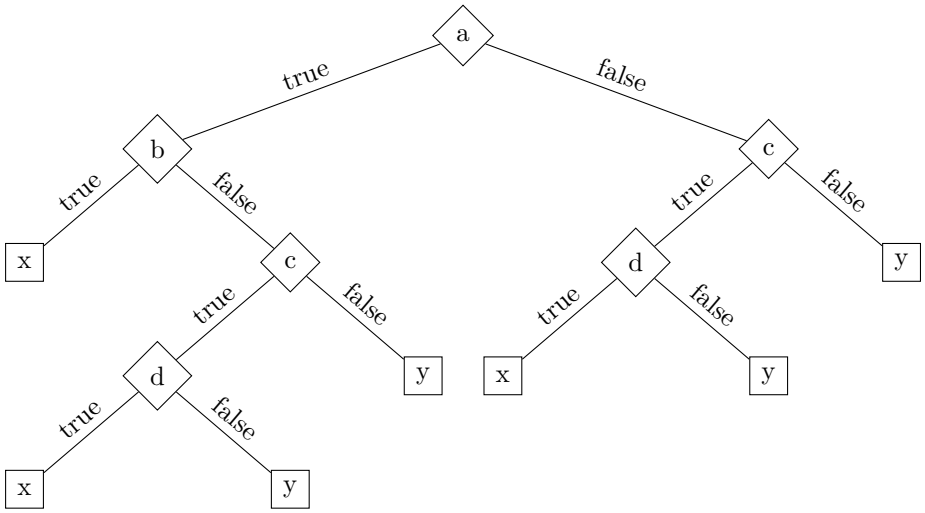


Figure 1.1: Decision Tree - Describing a simple disjunction [13]

Figure 1.1 illustrates a simple decision tree, also shown in [13]. However, its outcome may be a little misleading as it is a simple disjunction, which can be easily described using rules:

$$\begin{aligned} &if\ a \wedge b\ then\ x \\ else\ &if\ c \wedge d\ then\ x \\ &else\ y \end{aligned} \tag{1.10}$$

In his work [17], Quinlan designed an algorithm called C4.5rules, which converts a decision tree to a decision list. After its construction, it tries to improve it. Unfortunately, this part of the algorithm is expensive. Cohen [19] showed that the complexity is near $\mathcal{O}(n^3)$, where n is the number of samples.

1.4 Rule Learning Algorithms

This section describes various methods for creating rule-based classifiers. Probably the simplest one would be to identify a feature that describes a given set of examples the best and uses it for further classification. This idea was first implemented in 1993 by Holte [20] and is called 1R, or 1-rule.

1.4.1 1R

According to Holte [20], there were already some indications that simple rules could perform well on already known datasets. This turned out to be accurate, and Holte's 1R stayed behind Quinlan's C4.5 [17], a state-of-the-art algorithm used to create a decision tree model, by 2.1% points on average [20].

Describing the Steps of 1R

1R takes the train set as an input and examines its features. At the very beginning, 1R creates an empty hypothesis (step 1), which is later used to find the best performing decision list R. Then, it iterates through all features in the train set (step 2), and for each step, an empty decision list R is created (step 3). For every value of a given feature (step 4), 1R calculates how often each class appears (step 5) and creates a specific rule that best describes the most frequent class. If multiple classes satisfy this condition, a random class is selected. The relation $f \odot v$ depends on the implementation; for example, we could use the operator $=$ for categorical features or $<$ for numeric features (step 6). $\rightarrow C$ denotes that the rule assigns the samples to class C. This is specific for 1R as it can alternate between different classes with its decision list. The rule is inserted into decision list R (step 7). After iterating through all values, an error rate for decision list R is calculated (step 9). This decision list is inserted into hypothesis H (step 10). Finally, a decision list R with the lowest error rate is found (step 12) and returned.

Algorithm 1: 1R

Input: Train set
Output: Decision list

$H \leftarrow \{\}$; `// hypothesis`
for $f \in features$ **do**
  $R \leftarrow [\,]$;
  for $v \in f$ **do** `// for every feature's value` v
    $C \leftarrow$ most frequent class;
    create rule r such that $f \odot v \rightarrow C$;
    $R \leftarrow R \cup r$; `// insert r into R`
  end
  $err \leftarrow R\, error\, rate$;
  $H[R] \leftarrow err$
end
find $R \in H$ such that its err is lowest;
return R;

However, we have not mentioned how 1R deals with continuous and missing values. Missing values are simply treated as another value, e.g., "missing". Continuous values are divided into several intervals. To avoid overfitting, each interval must contain at least 3 or 6 samples of the same class. These specific numbers were experimentally obtained by Holte et al. in one of his previous papers [20].

If applied to the samples in Table 1.1, we would get the results shown in Table 1.2. We can see that 1R found such a decision list that classifies all samples from the train set correctly. The resulting list would have the following form:

$$\begin{aligned} &if\ printables\ \# < 8479\ then\ benign \\ &elif\ printables\ \# \geq 8479 \wedge printables\ \# < 25774\ then\ malware \\ &elif\ printables\ \# \geq 25774\ then\ benign. \end{aligned} \tag{1.11}$$

1.4.2 CN2

Many rule learning algorithms learn iteratively one rule at a time while removing the samples they cover from the train set [21]. Such an approach is called sequential covering and was also adapted by Clark and Niblett in their CN2 algorithm [22], inspired by Quinlan's well-known ID3 [23] and Michalski's A^q [24] algorithms. Both ID3 and A^q assume there is no noise present in the domain and search for a perfect description of the train data. However, when dealing with real-world data, we expect some noise to be present in the domain and need to find a way to handle it.

Table 1.2: Evaluating samples in Table 1.1 with 1R

feature	rule	errors	total errors
filesize	$< 156{,}784 \rightarrow$ benign	0/3	2/10
	$\geq 156{,}784 \wedge < 773{,}044 \rightarrow$ malware	1/4	
	$\geq 773{,}044 \wedge < 966{,}595 \rightarrow$ benign	1/3	
size of headers	$< 4{,}096 \rightarrow$ benign	3/8	4/10
	$\geq 4{,}096 \rightarrow$ malware	1/2	
timestamp	$<$ 2016-10-11 $\rightarrow$ benign	2/5	3/10
	$\geq$ 2016-10-11 $\wedge <$ 2018-5-29 $\rightarrow$ malware	1/3	
	$\geq$ 2018-5-29 $\rightarrow$ benign	0/2	
printables #	$< 8{,}479 \rightarrow$ benign	0/4	0/10
	$\geq 8{,}479 \wedge < 25{,}774 \rightarrow$ malware	0/4	
	$\geq 25{,}774 \rightarrow$ benign	0/2	

Clark and Niblett presented three requirements that should be met by rule learning algorithms to be considered useful in the real world domain. They should be accurate even for noisy data, efficient in rule generation, and the result should be simple enough to remain well comprehensible. The algorithm is shown in Algorithm 2.

Algorithm 2: CN2

Input: Train set – $TSet$
Output: Decision list

$R \leftarrow \{\}$;
repeat
  $r \leftarrow Find\_Best\_Rule(TSet)$;
  if *r is not empty* **then**
    $TSub \leftarrow coverage(r, TSet)$;
    $TSet \leftarrow TSet \backslash TSub$;
    $C \leftarrow$ most frequent class in $TSub$;
    let $r \rightarrow C$;
    $R \leftarrow R \cup r$;
  end
until *$TSet$ is empty or r is empty*;
return R;

Describing the Steps of CN2

From a high point of view, CN2 is not very complicated and corresponds to the already mentioned description of sequential covering. It starts with an empty decision list (step 1) and tries to cover as many samples from the train set as possible. This process ends once the train set is empty or no further rules can be found. In each iteration, CN2 calls its other function (step 3), *Find_Best_Rule*, presented in Algorithm 3, which returns the rule that describes the train set the best at a given moment. If the rule is not empty (step 4), then all samples covered by the rule are found and inserted into $TSub$ (step 5). These samples are then removed from the train set (step 6), and the most frequent class for $TSub$ is found (step 7). This class is declared to be the consequent of the rule (step 8), and the rule is then inserted into the decision list with respect to order (step 9). Finally, decision list R is returned (step 12).

Algorithm 3: Find_Best_Rule

Input: Train set – $TSet$, maximum rule size – max_size
Output: best rule r

1 $S \leftarrow \{\}$;
2 $best\_r \leftarrow empty$;
3 $conditions \leftarrow \{\, c \mid every\ possible\ c = feature \odot value\}$;
4 **repeat**
5 $\quad N \leftarrow \{\, r \wedge c \mid r \in S, c \in conditions\}$;
6 $\quad N \leftarrow N \backslash (S \cup \{\, r \mid r \in N, r\ contradicts\ itself\})$;
7 $\quad$ **foreach** $r \in N$ **do**
8 $\quad\quad$ **if** r *is statistically significant* $\wedge$ *usr_fun(r,TSet)* **then**
9 $\quad\quad\quad best\_r \leftarrow r$;
10 $\quad\quad$ **end**
11 $\quad$ **end**
12 $\quad$ **repeat**
13 $\quad\quad r \leftarrow worst\ rule\ in\ N$;
14 $\quad\quad N \leftarrow N \backslash r$;
15 $\quad$ **until** $|N| \leq max\_size$;
16 $\quad S \leftarrow N$;
17 **until** S *is empty*;
18 return $best\_r$;

Describing the Steps of Find_Best_Rule

Find_Best_Rule examines the train set and tries to find a rule that best describes it in its current state. The original algorithm used variable names $STAR$ and $NEWSTAR$, which we refer to as S and N for the sake of readabil-

ity. The function starts with an empty decision list S (step 1) and an empty rule (step 2), $best\_r$. A set of conditions is created for every feature and every possible value. Then in each step, until S is empty, a set N is created. The relation in $f \odot v$ depends on the implementation; however, if more relations apply to one feature, all should be included (step 3). N is built by combining the rules in S and the conditions set (step 5). Rules already contained in S and rules that contradict themselves are removed from N (step 6). Every rule created this way is checked whether it is statistically significant and approved by a user-defined function (step 8). If so, the best rule so far is replaced by any rule performing better than it on the train set (step 9). Then, until the size of N is lower than the user-defined maximum (step 15), rules that perform the worst are removed from N (step 13–14). S is then replaced by N. As a result, the best rule is returned (step 18).

Both the significance of the rules and usr_fun are heuristics used in CN2 to determine the quality of the rules. To test the significance of the rules, Clark and Niblett [22] used the likelihood ratio statistic [25] given by the following definition.

Definition 7 (Likelihood Ratio Statistic)**.** *Likelihood Ratio Statistic is defined by*

$$2 \sum_{i=1}^{n} f_i \log_2 \frac{f_i}{t_i}, \tag{1.12}$$

where n is the number of classes, f_i is the class frequency with respect to the decision list and t_i is the real class frequency.

This is done under the assumption that the rules select samples randomly.

The second heuristic is a user-defined function, usr_fun. In their paper [22], Clark and Niblett used the information-theoretic entropy for function usr_fun.

Definition 8 (Entropy)**.** *Entropy is defined by*

$$-\sum_{i=1}^{n} p_i \log_2 p_i, \tag{1.13}$$

where n is the number of classes and p_i the probability for a given class.

In this case, the smaller the entropy, the better the result.

When dealing with continuous attributes, CN2 divides them into discrete subranges. The paper does not mention the exact process of this discretisation. Missing feature values are replaced by the most common value occurring for that feature, respectively, the mean in the case of continuous features [22].

1.4.3 I-REP

A standard way of dealing with noisy data in decision trees is called pruning. A decision tree model that perfectly describes the train set is generated. To avoid overfitting, branches are then cut off according to a given criterion. In 1994, Fürnkranz and Widmer introduced an algorithm called Incremental Reduced Error Pruning, **I-REP** [26], which implements two pruning approaches: pre-pruning and post-pruning, corresponding to the already described pruning process for decision trees. Pre-pruning ignores some training examples so that the final concept would not describe the train set perfectly.

Fürnkranz and Widmer compare their approach to I-REP's predecessor, Reduced Error Pruning, **REP**. REP uses post-pruning to avoid noise that could be potentially present in the train set. Although this technique is quite effective in noisy domains, it also has several disadvantages.

For a large set of data, REP turns out to be very inefficient, as its complexity is $\mathcal{O}(n^4)$, where n is the number of samples in the train set. I-REP is by several magnitudes faster, and its complexity is $\mathcal{O}(n\,log^2(n))$. Both I-REP and REP split the train data into two sets, a growing set and a pruning set. The growing set is used to learn new rules and the pruning set allows for the rules to stay more general. The difference is that REP creates both of these sets only once, and thus the pruning set is used for post-pruning only. This means that REP heavily depends on how both sets are created. On the other hand, I-REP splits examples during each iteration and reduces REP's problem to a single rule learning. This strategy has drawbacks as it may consider a nonempty rule worse than an empty one.

In contrast to decision trees, which use the divide-and-conquer technique, rule learning algorithms use the separate-and-conquer technique. Both techniques are relatively similar, but separate-and-conquer first focuses on the part of the train set and tries to describe it. In contrast, divide-and-conquer strives to maximise the separation between classes [13]. Fürnkranz and Widmer pointed out another significant difference: branch pruning does not affect other branches, while rule pruning does affect other rules. This way, REP does not have to pick the correct features, and pruning will not necessarily help. I-REP solves this by pruning the rule right after it has been learned and by using a decision list to preserve the order.

The last disadvantage mentioned in the paper is using a bottom-up hill-climbing technique to maximise the accuracy. In noisy domains, the rules can get too specific, resulting in a lot of pruning and a higher chance of getting caught in a local maximum. Instead of bottom-up hill-climbing, I-REP uses a top-down approach. This means that the final result is found by gradually adding rules to an empty decision list.

Algorithm 4: I-REP

Input: Positive samples – Pos, Negative samples – Neg, $SplitRatio$
Output: Decision list R

1 $R \leftarrow \{\}$;
2 **while** $Pos \neq \emptyset$ **do**
3 $\quad SplitExamples(SplitRatio, Pos, PosGrow, PosPrune)$;
4 $\quad SplitExamples(SplitRatio, Neg, NegGrow, NegPrune)$;
5 $\quad r \leftarrow empty\ rule$;
6 $\quad$ **while** $NegGrow \neq \emptyset$ **do**
7 $\quad\quad r \leftarrow r \wedge FindLiteral(r, PosGrow, NegGrow)$;
8 $\quad\quad PosGrow \leftarrow coverage(r, PosGrow)$;
9 $\quad\quad NegGrow \leftarrow coverage(r, NegGrow)$;
10 $\quad$ **end**
11 $\quad r \leftarrow PruneRule(r, PosPrune, NegPrune)$;
12 $\quad$ **if** $Accuracy(r) \leq Accuracy(fail)$ **then**
13 $\quad\quad$ return R;
14 $\quad$ **end**
15 $\quad$ **else**
16 $\quad\quad Pos \leftarrow Pos \backslash coverage(\mathrm{r}, Pos)$;
17 $\quad\quad Neg \leftarrow Neg \backslash coverage(\mathrm{r}, Neg)$;
18 $\quad\quad R \leftarrow R \cup r$;
19 $\quad$ **end**
20 **end**
21 return R;

Describing the Steps of I-REP

I-REP requires the data to be split into positive and negative samples in its input. One can also specify a split ratio, which is by default set to $\frac{2}{3}$. I-REP starts with an empty decision list (step 1) and completes it by iteratively adding rules until the set of positive samples, Pos, is empty (step 2). Both Pos and Neg are randomly split into a growing set (2/3) and a pruning set (1/3) (step 3–4). Then, the algorithm tries to find such a rule that does not cover the negative growing set (step 6–10). However, the paper does not further specify how this should be done. We could argue that for some sets, this condition will never be satisfied. In other words, we have no guarantee that a rule that covers the positive growing set the best will not cover the negative growing set. Thus, this part of the code can get stuck in an infinite loop, and another condition is necessary to break out of it. After the rule is built, it gets pruned using both pruning sets (step 11). *PruneRule* removes the last condition to maximise pruning metrics mentioned below. The accuracy of this rule must not be lower than the accuracy of an empty rule given by

$\frac{N}{P+N}$ (step 12). If so, the algorithm ends and returns decision list R (step 13). Otherwise, it removes the covered samples from both positive and negative sets (steps 16–17), and it appends the rule to the decision list with respect to order (step 18).

The paper mentions two variants of I-REP. Both differ in the metrics that are used for pruning. I-REP-1 tries to maximise the following formula:

$$\mathcal{P}_{\text{I-REP-1}}(p, P, n, N) = \frac{p + (N - n)}{P + N}, \tag{1.14}$$

where p (n) is the number of positive (negative) samples covered by the current rule from a total number of P (N) positive (negative) samples in the pruning set. The accuracy of the rules must not fall under $\frac{N}{P+N}$. I-REP-2 maximises

$$\mathcal{P}_{\text{I-REP-2}}(p, P, n, N) = \frac{p}{n + p}, \tag{1.15}$$

and the accuracy, also given by this formula, must not fall under 50%.

I-REP is an algorithm well suited for two-class classification. At the time I-REP's implementation was not ready for multiclass classification, neither was it capable of handling continuous attributes. We could run the algorithm for every class and divide them into discretised intervals in the case of continuous attributes. The paper does not mention how it would handle missing values, too.

1.4.4 RIPPER

Thanks to its complexity, I-REP is very efficient on large train sets. However, in 1995, Cohen showed that I-REP does not learn rules well enough and can be outperformed by previously known algorithms, such as C4.5rules [19]. Cohen has addressed specific issues and explained how I-REP could be improved.

In his paper [19], Cohen introduced a new algorithm called "Repeated Incremental Pruning to Produce Error Reduction", **RIPPER**, a direct successor of I-REP. Initially, Cohen's team was only interested in replicating Fürnkranz's and Widmer's results and in doing further testing.

Cohen's implementation of I-REP differs in some parts. First, when growing a rule, FOIL's information gain [27] is maximised.

Definition 9 (FOIL's information gain)**.** *We define FOIL's information gain of rules r_0 and r_1 as follows:*

$$\mathcal{F}(r_0, r_1) = p_0(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0}), \tag{1.16}$$

where p_0 (n_0) is the number of positive (negative) samples covered by rule r_0, and p_1 (n_1) is the number of positive (negative) samples covered by rule r_1.

Their implementation allows multiple conditions to be deleted in pruning, whereas the original implementation allowed only one condition to be deleted. Rule learning stops when a newly generated rule has an error rate greater than 50%.

Cohen and his team also generalised the algorithm for multiclass problems. The classes are sorted in ascending order based on their prevalence; for example, the least frequent class is the first, and the most frequent class is the last. Then, for each class, I-REP is used to separate it from other classes. Their implementation can handle continuous features as well as missing values. When testing a sample with a missing value, the test fails.

To improve the algorithm, Cohen's team made three modifications. Based on their testing, they traced that the original Metric (1.14) used for pruning (assuming P and N are fixed) was significant as to why the algorithm has not been able to converge in some cases. As shown by Cohen, this metric prefers rule r_i that covers $p_i = 2000$ positive samples and $n_i = 1000$ negative samples to rule r_j that covers $p_j = 1000$ positive samples and $n_j = 1$ negative sample. The metric was then replaced by

$$\mathcal{P}_{\text{RIPPER}}(p, P, n, N) = \frac{p - n}{p + n}. \tag{1.17}$$

I-REP's rule learning often stops too soon due to the condition of the maximal error rate of 50%. The solution to this was to minimise the total description length instead [19, 17].

Definition 10 (Description Length)**.** *Given the positive real numbers n, k and $p \neq 1$, we define the description length as follows:*

$$\mathcal{S}(n, k, p) = \frac{1}{2}(k \log_2 \frac{1}{p} + (n - k) \log_2 \frac{1}{1 - p} + \log_2 k), \tag{1.18}$$

As described by Cohen [19], this encoding allows two parties (sender and recipient) to work over a set of n elements. The recipient can recognize k elements, and p is known ahead. $\log_2 k$ is the number of bits required to send the number k. The whole metric is scaled by $\frac{1}{2}$ to limit possible redundancy in the features.

Definition 11 (Decision List Exceptions)**.** *For a given set of samples S with a positive class P and a negative class N, and for a given decision list R, we define the number of exceptions as follows:*

$$\mathcal{E}(R, S) = \log_2 \binom{tp}{fp} + \log_2 \binom{tn}{fn}, \tag{1.19}$$

where tp (tn) is the number of samples correctly classified as P (N), and fp (fn) is the number of samples incorrectly classified as P (N).

Definition 12 (Total Description Length)**.** *For a given set of samples S and a decision list R we define its total description length as follows:*

$$\mathcal{T}(R,S) = \sum_{\substack{r \in R \\ \text{in order}}} \mathcal{S}(n, k_r, \frac{k_r}{n}) + \mathcal{E}(R,S), \tag{1.20}$$

where n is the total number of possible conditions for S and k_r is rule r's length.

Learning stops once the total description length is more than d bits larger than the smallest total description length found so far, or if there are not any positive samples remaining. Cohen and his team used $d = 64$ for their experiments.

The last step to improve I-REP's performance was to add rule optimisation. For each rule, in the order they have been learned, two new rules are created. The first rule, called replacement, is created by growing an empty conjunction. The second rule, called revision, is grown by adding conditions to the original rule. Both rules are pruned and minimise Metric (1.14) on the whole pruning set. At the end of this process, it has to be decided whether the replacement, the revision, or the original rule should be used. Out of the three, the rule that lowers the total description length the most is picked.

Describing the Steps of I-REP*

RIPPER can be divided into two parts, I-REP*, see Algorithm 5, which similarly grows a decision list as I-REP does, and the optimisation part in Algorithm 6. Unlike I-REP, I-REP* can start with a nonempty decision list, which is further expanded. For steps 2–6 and steps 14–16, see the Algorithm 4. After growing and pruning a rule, the total description length (**TDL**) (see Definition 12) is calculated (step 7). Then, if TDL is better than the minimum description length (**MDL**), which is by default set to $+\infty$ (step 1), TDL replaces MDL (steps 8–10). Else, if TDL is greater than MDL by more than d bits, the loop breaks (steps 11–13). Then for each rule in reversed order, check whether or not the rule increases MDL. If it does, remove it (steps 18–24). Note, however, that the original paper [19] does not specify how to minimise MDL or in what order should be the rules removed.

Describing the Steps of RIPPER

The second part of the algorithm RIPPER starts by calling I-REP* to build a new decision list R (step 1). k iterations follow, by default two (step 2). To keep track of the remaining samples, $RPos$ and $RNeg$ are initialised. Then, for every rule r in decision list R, in the order in which they were learned, the samples are split into a growing set and a pruning set (steps 6–7). Two new rules are created, rev and rep. The process for both is quite similar, yet

Algorithm 5: I-REP*

Input: Positive samples – Pos, Negative samples – Neg, $SplitRatio$, Largest bit difference – d, Decision list – R
Output: Decision list R

```
1  MDL ← +∞;
2  while Pos ≠ ∅ do
3  |  SplitExamples(SplitRatio, Pos, PosGrow, PosPrune);
4  |  SplitExamples(SplitRatio, Neg, NegGrow, NegPrune);
5  |  r ← GrowRule(PosGrow, NegGrow);
6  |  r ← PruneRule(r, PosPrune, NegPrune);
7  |  TDL ← total_description_length(R);
8  |  if TDL < MDL then
9  |  |  MDL ← TDL
10 |  end
11 |  else if TDL − MDL > d then
12 |  |  break
13 |  end
14 |  Pos ← Pos \ coverage(r, Pos);
15 |  Neg ← Neg \ coverage(r, Neg);
16 |  R ← R ∪ r;
17 end
18 for r_i, i ∈ {|R|, ..., 1} do
19 |  MDL ← total_description_length(R);
20 |  TDL ← total_description_length(R \ r_i);
21 |  if TDL < MDL then
22 |  |  R ← R \ r_i;
23 |  end
24 end
25 return R;
```

revise starts by growing an empty conjunction (step 8) and replace starts by adding new conditions to the iterated rule r (step 11). Both rules are then pruned, and two new decision lists are constructed, *Revise* and *Replace*, both without the rule r, but containing the new rules *rev* and *rep* (steps 9–10 and steps 12–13). Decision list that minimises TDL is the new decision list R (step 15). The rule r is updated accordingly (step 16); it either stays the same or becomes *rev* or *rep*. Samples that are covered by the rule r are removed from $RPos$ and $RNeg$ (steps 17–18). To cover any remaining samples, I-REP* is called again with the existing decision list R.

Algorithm 6: RIPPER

Input: Positive samples – Pos, Negative samples – Neg, $SplitRatio$, Largest bit difference – d, Number of iterations – k
Output: Decision list R

$R \leftarrow I\text{-}REP*(Pos, Neg, SplitRatio, d, \{\})$;
for $1 \dots k$ **do**
  $RPos \leftarrow Pos$; `// remaining positive samples`
  $RNeg \leftarrow Neg$; `// remaining negative samples`
  for $r \in R$ **do**
    $SplitExamples(SplitRatio, RPos, PosGrow, PosPrune)$;
    $SplitExamples(SplitRatio, RNeg, NegGrow, NegPrune)$;
    $rev \leftarrow GrowRule(PosGrow, NegGrow)$;
    $rev \leftarrow PruneRule(rev, PosPrune, NegPrune)$;
    $Revise \leftarrow (R \setminus r) \cup rev$;
    $rep \leftarrow GrowRule(PosGrow, NegGrow, r)$;
    $rep \leftarrow PruneRule(rep, PosPrune, NegPrune)$;
    $Replace \leftarrow (R \setminus r) \cup rep$;
    $DLs \leftarrow \{R, Revise, Replace\}$; `// DLs ...Decision lists`
    $R \leftarrow argmin(DLs)$; `// minimise description length`
    update r ; `// based on the decision list` R
    $RPos \leftarrow RPos \setminus coverage(r, RPos)$;
    $RNeg \leftarrow RNeg \setminus coverage(r, RNeg)$;
  end
  $R \leftarrow I\text{-}REP*(RPos, RNeg, SplitRatio, d, R)$;
end
return R;

1.4.5 Other Approaches

In 2003, Dain et al. [28] published the algorithm I-REP++. It uses ideas proposed by Cohen [19] to improve I-REP's accuracy. The authors proved that FOIL's information gain is sortable. This allowed them to use special data structures that profit from this. Since they aimed to reduce time complexity mainly, they did not use the total description length (see Definition 12) to stop rule learning. Instead, I-REP++ stops after it has learned five bad rules (covers more negative than positive samples).

Rule learning search space can be of considerable size. That is why other rule learning methods, such as genetic algorithms, can be used in this domain. In [29], chromosomes (genes in a chromosome correspond to conditions – see Definition 1) are represented by classification rules (see Definition 2) with three mutation operators. Mutation operators change certain parts of the gene (e.g.

relation >= to <). Fidelis et al. have also used two-point crossover, which randomly exchanges two genes of two parental chromosomes [30]. Mutation operators and crossover are used to generate new offsprings. Offsprings are subject to fitness maximisation.

We have mentioned an indirect method for rule extraction from decision trees – C4.5rules. Generally, we could consider trained machine learning models as an oracle and use them to generate new data for rule learning. Craven and Shavlik [31] used this approach in their TREPAN algorithm. However, it is used to extract decision trees, not decision rules. Also, their oracle decides the class labels and splits for non-leaf nodes.

Chapter 2

Portable Executable

Portable Executable (**PE**) is an executable file format used by 32-bit and 64-bit Microsoft operating systems. Its first appearance dates back to 1993 with the release of the operating system Windows NT 3.1. Although there are some slight differences, the file format remains very similar to the original one. Note that 32-bit and 64-bit versions differ, too. This chapter will briefly describe the PE file format content, allowing the Windows operating system to map the executable into its memory and load it [32, 33]. The structure of the PE file format can be seen in Table 2.1.

2.1 Dos Header and Stub

To remain compatible with the Windows operating systems' previous versions, Microsoft has kept the original Disk Operating System header, shortly, **MS-DOS** header. The MS-DOS header is 64 bytes long and contains a pointer, commonly known as `e_lfanew`, to the PE File signature's file address [34].

Table 2.1: PE File Format

MS-DOS Header
MS-DOS Stub
PE File Signature
PE Header
Optional Header
Section Headers
Section Data

A short program also follows the header, referred to as STUB, which outputs "This program cannot be run in DOS mode." [35].

2.2 PE Header

The PE file header follows the PE file signature. It was inspired by the Common Object File Format (**COFF**) used by Unix. This header contains information about the target machine, the number of sections, the timestamp of the file linkage, and more [36].

2.3 Optional Header

Even though this header's name contains optional, it is only optional for the object files. This header is essential for executables and Dynamically Linked Libraries (**DLL**s). The optional header consists of helpful information about the executable, such as the initial stack size, program entry point, operating system version, and more. It also includes important details such as the import and export symbols, which can be accessed through the data directories [37].

2.4 Section Headers

The Section headers are straight away located after the Optional header. Each section header has its name, addresses, and a series of flags. The number of sections is not limited. Some of the typical headers are [38]:

- .text – combined code segments
- .data, .rdata,.bss – data variables, read only data, etc.
- .rsrc – resource information
- .edata – export section for an application/DLL
- .idata – import section, necessary for the loader to find the target functions

Chapter 3

Implementations of Rule Based Classifiers

Below we describe the specifics of our implementations of some of the algorithms mentioned in Section 1.4 and an additional algorithm that performs a different growing approach with an early stop condition. We have not implemented the CN2 algorithm as the worst time complexity of $Find\_Best\_Rule$ is $\mathcal{O}(f^2\, s^2\, n)$, where f is the number of features, s is the number of samples and n is the maximum size of set STAR (see Algorithm 3) [22]. We have chosen the Python programming language for the implementations. Python can often serve as an easy to use extension of libraries written in C/C++ or Fortran, such as those available from the `SciPy` ecosystem [39]. We have used some of the vectorised operations[1] available in `NumPy` [41] to speed up the execution time. This approach is also recommended in the `scikit-learn` documentation [42]. We have also implemented a decision list class to reduce code redundancy.

3.1 Decision List

Decision list serves as a wrapper for built-in Python data structures. Each rule contained in the decision list class corresponds to Definition 1, where the relation may be one of the following operators: $\{<=, >=, in\}$. Operator $<=$ and operator $>=$ are intended to be used for numerical features and operator in for categorical features.

[1] NumPy's vectorised operations are efficiently implemented in C/C++ or Fortran. They are more compact, less error-prone, and often resemble well-known mathematical operations [40].

Definition 13 (Operator *in*)**.** *The operator in is defined as follows*

$$f \text{ in } s \equiv f = s_1 \vee \ldots \vee f = s_n, \tag{3.1}$$

where n is the number of values in the set $s = \{s_1, \ldots, s_n\}$.

3.2 1R

Our 1R implementation follows the steps in Algorithm 1. We have iteratively searched for splits with at least three members of one class to discretise numeric features. This fact is captured by $1R_{min\_cl\,=\,3}$ in Chapter 4. Note that in the case of 1R, we have not found much of a use of `NumPy`'s vectorised operations. The implementation is inefficient for larger data sets, and we only use it for comparison.

3.3 I-REP

As the original paper for I-REP (see Algorithm 4) does not cover how to deal with numerical features, we have used Cohen's [19] suggestions for the algorithm. During the growth phase, the algorithm searches for the best split between the numerical and the categorical features. Rule growth for both I-REP and RIPPER is guided by maximising FOIL's gain and stops once there are no negative samples left in the growing set. Neither of the papers [26, 19] tackles the issue of learning the same rule twice. This may happen if present feature values are the same for both positive and negative growing sets. Our implementation stops the growing phase and proceeds to the next step.

Once a new rule is learned, it is immediately pruned. Pruning is guided by maximising Metric (1.14). When pruning, we start by removing the last condition learned. If the new rule created this way would increase Metric (1.14), we continue removing conditions that precede it. Else we stop the rule pruning and return the pruned rule.

We stop adding new rules if the last pruned rule should have an error rate higher than 50%.

3.4 RIPPER

In Section 1.4.4, we have mentioned that RIPPER can be divided into two separate algorithms – I-REP* and RIPPER (see Algorithms 5 and 6). We have described the rule growth and pruning phases in section 3.3 above. Unlike I-REP, I-REP* uses Metric (1.17) for pruning and uses different stop condition. The stop condition is guided by minimising the total description length (see Definition 12). Unfortunately, this metric is quite expensive, and we try to minimise its cost by caching rule description lengths (see Definition 10). In

Section 25, we have mentioned that Cohen [19] does not specify how to remove rules. We tried two additional approaches – Algorithm 7 and Algorithm 8 – that replace steps 18–23 in Algorithm 5 (see Section 4.4.2).

Algorithm 7: RIPPERk$_{\text{remove_all}}$

```
1 MDL ← total_description_length(R);
2 remove ← {};
3 for r_i, i ∈ {|R|, ..., 1} do
4 |   TDL ← total_description_length(R \ r_i);
5 |   if TDL < MDL then
6 |   |   remove ← remove ∪ r_i;
7 |   end
8 end
9 R ← R \ remove;
```

Algorithm 8: RIPPERk$_{\text{sort}}$

```
1 while ∃r ∈ R that increases TDL do
2 |   r ← r that increases TDL the most;
3 |   R ← R \ r;
4 end
```

In Algorithm 7, any rule that increases TDL is added to set *remove*, which is then removed from decision list R. Algorithm 8 does not stop removing rules until no rules remain that increase TDL. Both approaches are less efficient than the one used in Algorithm 5.

We followed the steps in Algorithm 6 during RIPPER's optimisation phase. We maximise Metric (1.14) on the whole pruning set in the pruning phase. RIPPER can have k iterations; we note this fact by RIPPERk.

3.5 Greedy Grow

We propose an algorithm that would not require the data to be divided into a growing and a pruning set during our experiments. By doing so, we believe that we can consider all the data, and if any noise should be present, it should be avoided by carefully chosen metrics.

Naturally, since we have seen I-REP and RIPPER, a question arises, how are we supposed to prune our rules? We could use the same data we used for growing for pruning, or we could not use any pruning at all. However, this could lead us down a dangerous path, as we could easily overfit the data and produce lengthy rules. This may lead to similar problems as described in PRISM [18]; that is, the rules could potentially use features that are irrelevant

to our problem. To prevent this, we could either stop condition search or rule growth earlier. We tried the first approach in Algorithm 9, described below.

Algorithm 9: Greedy Grow

Input: Positive samples - Pos, Negative samples - Neg, Stop Grow - $stop_{search}$, Stop Learn - $stop_{learn}$
Output: Decision List R

```
1  R ← {};
2  while Pos ≠ ∅ do
3  |  PosGrow, NegGrow ← Pos, Neg;
4  |  r ← {};
5  |  while NegGrow ≠ ∅ do
6  |  |  PosS, NegS ← |PosGrow|, |NegGrow|;
7  |  |  while True do
8  |  |  |  c ← FindLiteral(r, PosGrow, NegGrow, PosS, NegS);
9  |  |  |  if r ∪ c = r ∨ stop_search(r ∪ c) then
10 |  |  |  |  break
11 |  |  |  end
12 |  |  |  r ← r ∪ c;
13 |  |  |  PosGrow ← coverage(r, PosGrow);
14 |  |  |  NegGrow ← coverage(r, NegGrow);
15 |  |  end
16 |  |  if r has not changed then break ;
17 |  end
18 |  if stop_learn(r, Pos, Neg) then
19 |  |  break;
20 |  end
21 |  Pos ← Pos \ coverage(r, Pos);
22 |  Neg ← Neg \ coverage(r, Neg);
23 |  R ← R ∪ r;
24 end
25 return R;
```

In addition to positive and negative samples, we can also specify function $stop_{search}$, function $stop_{learn}$ to identify when to stop searching, learning. In our version of the algorithm, we stop the condition search if the number of negative samples should decrease by less than 20% of the current negative sample size ($NegS$). We ran several experiments, and 20% seems to be producing accurate and more compact decision lists than a lower percentage. We note this by $\text{GREEDY}_{20\%}$. We have also decided to stop learning in the same way that I-REP* does, that is, by minimising TDL (see steps 7–13 in Algorithm 5). We need to ensure that the generated rule is not an empty conjunction when stopping the condition search.

After initialising $PosGrow$ and $NegGrow$, two nested loops follow (step 3). The first one can be understood as rule growth (step 5) and the second one as condition search (step 7). When searching for literals, we are maximising the following metric:

$$\mathcal{G}_{GREEDY}(p, P, n, N) = \frac{p}{P}(1 - \frac{n}{N}), \tag{3.2}$$

where p (n) is the number of positive (negative) samples covered by the rule r, and P (N) is the total number of positive (negative) samples. Using this metric, we make no distinction between a rule that covers 80% positive samples and 70% negative samples and a rule that covers 25% positive samples and 4% negative samples. This allows identifying potentially good rules quickly. Note that $FindLiteral$ takes as an input $PosS$ and $NegS$, too (step 8). They correspond to the total number of positive (negative) samples in Metric (3.2). If adding a condition c to the rule r does not change it or $stop_{search}$ fires, we stop the condition search (steps 9–11). Otherwise, we add the condition to the rule (step 12) and proceed with the condition search. We only check whether the rule has changed or not in the outer loop (step 16). After checking $stop_{learn}$, we remove samples covered by the rule r and add the rule r to R (steps 21–23).

Chapter 4

Experiments

Our aim throughout the experiments was to find out how well do rule-based classifiers (**RBC**s) interpret the results of various machine learning methods. Along with these experiments, we were also interested in behavioural changes of the algorithms should we replace pruning metrics. Additionally, we tested other rule removing approaches mentioned in Section 3.4. We used the Python programming language as we did for the implementations, along with the interactive environment Jupyter Notebook [43], packages from the `SciPy` ecosystem [39], or modules in the Python library (e.g. `pickle` [44] to save experiments' outcomes). The implementations of the RBCs and the experiments can be found on the enclosed SD card.

4.1 Dataset Description

We have collected two datasets. A smaller one has been made available to us by Kozák [45], whom we thank for access, and the other one called **EMBER** (Elastic Malware Benchmark for Empowering Researchers) [16] is freely available to researchers.

4.1.1 Kozák's Dataset

Kozák's dataset consists of 30,154 samples and 303 columns. Columns are built from categorical and numerical features, such as flags, hashes or section entropy. One of the columns, `is_malware`, indicates whether the file is benign (0) or malicious (1). The data is divided into 27,473 malicious samples and 15,077 benign samples. More details about the features can be found in the file `feature_transformation.ipynb` on the enclosed SD card.

4.1.2 EMBER Dataset

The EMBER dataset consists of 1.1M samples, divided into a train set with 900K samples (300K malicious, 300K benign, 300K unlabeled) and a test set with 200k samples (100K malicious, 100k benign). The feature set is divided into eight groups of raw features. Similarly to Kozák's dataset, the EMBER dataset includes the header information, lists of imported functions, byte histograms, and more. We have ignored the unlabeled samples in the train set throughout our experiments.

4.2 Feature Transformation and Selection

Even though RBCs can handle numerical and categorical features, traditional machine learning algorithms and the implementations available from `scikit-learn` [46], which we used to train our models, require the features to be numerical only. Thus we performed similar steps to transform the datasets as in Kozák [47], briefly described below.

4.2.1 Feature Vectorisation

Kozák's dataset includes features (documents) that are sequences of strings, for example, that inform the program user about an error. Some strings may or may not carry a piece of important information as to whether the file is malware or not. To transform string features into a sparse matrix, we used a measure called **TF.IDF** (Term Frequency times Inverse Document Frequency) [48], which assign every string a number with the following properties.

Definition 14 (Term Frequency)**.** *The Term Frequency is defined as follows:*

$$TF_{ij} = \frac{f_{ij}}{\max_k f_{kj}}, \tag{4.1}$$

where f_{ij} is the frequency of string i in document j.

Definition 15 (Inverse Document Frequency)**.** *The Inverse Document Frequency is defined as follows:*

$$IDF_i = \log_2 \frac{N}{n_i}, \tag{4.2}$$

where N is the number of all documents and n_i is the number of occurences of string i across all documents.

The TF.IDF for string i in feature j is then simply defined as $TF_{ij} \times IDF_i$. Strings with the highest TF.IDF often offer a good description of a given feature [48]. `scikit-learn` implements this in `TfidfVectorizer` [49], which

we used for the transformation. Some of the words (strings) do not bear any meaning on their own even though they occur quite often, for example, "or" and "the". They are called stop words. Thus, we have supplied the `TfidfVectorizer` with the `stop_words` parameter for languages, which we have identified to be present in the features and the `max_df` parameter to exclude statistically uninteresting strings.

4.2.2 Feature Hashing

For the rest of the categorical features, we have used the class `FeatureHasher`, which is also implemented in `scikit-learn`. `FeatureHasher` [50] uses a technique called the hashing trick. Feature names are transformed into a sparse matrix, and a hash function is used to calculate the index of a categorical value.

The authors of the EMBER dataset published a code [51] that transforms available raw features into vectorised ones using the hashing trick. We have decided to use this code to transform the features and ended up with 2,381 new ones.

4.2.3 Feature Selection

After transforming features in Kozák's dataset, we immediately removed features with zero standard deviation and features that were duplicates only. The transformed dataset consisted of 1,290 features.

Before proceeding further, it was also necessary to standardise the data. Some machine learning algorithms' behaviour may worsen if the data do not appear to be from the normal distribution [52]. We have used the class `MinMaxScaler` from `scikit-learn` that transforms the features as follows:

$$x_{std} = \frac{x - \min(X)}{\max(X) - \min(X)}, \tag{4.3}$$

where x is the original value and X is the collection of every value in a given feature.

Consequently, we have decided to use yet another `scikit-learn` tool, `SelectFromModel`, to reduce the dimensionality of both datasets. This tool is a meta-transformer that accepts an estimator as a parameter [53]. The estimator must store feature importances. For example, feature importances for random forest classifier are given by two measures: how much does accuracy, Gini impurity, decrease if a variable is eliminated, chosen to split a node [54]. We can then limit the number of selected features by setting either `threshold` or `max_features`. In our case, we have set `max_features` to 100.

For Kozák's dataset, we have used Extra Trees classifier, and for the EMBER dataset, Random forest classifier. Random forest classifier is an ensemble model built from many decision trees (see Section 1.3). The final decision of

this model is driven by a majority vote [48]. The difference between the two is that random forest makes the input more diversified and chooses an optimum split on the selection of cut points. Extra Trees classifier works with the original input and chooses the split randomly [55]. We could also use other methods for feature selection, such as Principal Component Analysis (**PCA**) [56]. However, PCA transforms the features, which results in interpretability loss. Therefore we have decided to use the aforementioned methods.

In the end, we have identified features that carried no useful information as to whether the file is malicious or not. This was done using the 1R algorithm (see Algorithm 1), and we removed three features that had less than 10% of an error rate, e.g. feature `is_exe`.

4.3 Evaluation Metrics

To understand how well do machine learning (**ML**) algorithms or RBCs perform, we use several different metrics described in this section. We first define terms that are used in the metrics [57].

- True Positive (**TP**) – Correctly predicted malicious samples as malicious
- True Negative (**TN**) – Correctly predicted benign samples as benign
- False Positive (**FP**) – Incorrectly predicted benign samples as malicious
- False Negative (**FN**) – Incorrectly predicted malicious samples as benign

Using the terms above, we can calculate the false positive rate (**FPR**), also referred to as the fall-out rate [58], the true positive rate (**TPR**), also known as sensitivity and accuracy (**ACC**).

$$FPR \equiv \frac{FP}{FP + TN} \tag{4.4}$$

$$TPR \equiv \frac{TP}{TP + FN} \tag{4.5}$$

$$ACC \equiv \frac{TP + TN}{TP + TN + FP + FN} \tag{4.6}$$

To better distinguish between individual performances of RBCs, we use additional metric. We denote the number of rules in a decision list as **DL size** and the mean number of conditions for rules in the decision list as **ø r size**. We highlight those numbers that have the lowest "DL size × ø r size" throughout the experiments.

Table 4.1: Well known ML algorithms and their performance on Kozák's dataset, and the EMBER dataset

	Kozák's Dataset			EMBER Dataset		
ML algorithm	ACC	TPR	FPR	ACC	TPR	FPR
LogisticRegression	95.34	95.90	5.59	57.58	89.88	74.72
RandomForest	**98.82**	**99.43**	2.38	**89.21**	**92.01**	13.58
DecisionTree	98.66	98.89	2.18	88.86	90.20	**12.49**
kNN	98.64	98.52	**2.15**	78.86	77.97	20.26
AdaBoost	98.25	98.85	3.11	87.00	90.25	16.25

4.4 Interpreting ML results using RBCs

We have used five ML algorithms to classify the datasets for the purpose of our experiments – Logistic Regression (**LR**), Random Forest (**RF**), Decision Tree (**DT**), k-nearest neighbours (**kNN**) and AdaBoost. For Kozák's dataset, we have also fine-tuned the parameters, see file `ml_tuning.ipynb` on the enclosed SD card. To reduce the number of biases, the training was done using 5-fold cross-validation [48]. 5-fold cross-validation works as follows: the data is divided into five equal chunks. One of the five chunks is declared as the test data, and the rest of the chunks as the train data. This is repeated five times, each time with a different chunk for test data, and the final result is the average of all five results. The averaged results of each ML algorithm can be seen in Table 4.1. We have also saved the predictions on the test data during each fold.

Then, we used our RBCs implementations and trained them on ML algorithms' predictions. That is, for each ML algorithm and all of its five predictions on the test data, RBCs were used to describe the outcomes of that ML algorithm. The results were then averaged and are shown in Table 4.2.

We can see that 1R is easily outperformed by all other algorithms, except I-REP's FPR. Unfortunately, its simple approach leads to very long decision lists, which could still be understandable by machines, but not so much by human beings. Despite its high FPR and second-lowest accuracy, I-REP produces straightforward decision lists that could be easily understood by humans, too. Our algorithm, $\text{GREEDY}_{20\%}$, seems to be doing well in accuracy and FPR – on Kozák's dataset, it achieved the highest accuracy. However, it could still potentially be using irrelevant features, as its sum of all conditions is much greater than e.g. RIPPER's. Finally, we have also tested RIPPER with three different values for iterations (k). Although RIPPER does seem to increase its accuracy with each iteration, it does not increase its FPR, which seems rather unstable.

Table 4.2: Interpreting ML algorithms' results on Kozák's dataset using RBCs

time (s) shows the approximate time for training in seconds for each fold. For example, I-REP required 25 seconds in total for kNN.

	RBC	ACC	TPR	FPR	DL size	ø r size	time (s)
LR	$1R_{min\_cl=3}$	83.31	77.34	6.23	724.2	1.00	245
	I-REP	96.94	99.07	6.81	**5.4**	**2.97**	**5**
	RIPPER0	98.83	98.57	0.70	12.4	3.87	17
	RIPPER1	99.13	99.04	0.71	16.4	3.54	56
	RIPPER2	99.18	**99.11**	0.68	17.0	3.48	106
	$GREEDY_{20\%}$	**99.35**	98.98	**0.00**	17.60	5.64	30
RF	$1R_{min\_cl=3}$	80.18	74.16	8.74	2976.2	1.00	244
	I-REP	95.77	**99.58**	11.24	**6.6**	**2.84**	**5**
	RIPPER0	99.25	99.06	0.38	14.2	4.11	18
	RIPPER1	99.41	99.33	0.43	19.8	3.36	56
	RIPPER2	99.52	99.47	0.39	23.8	3.04	96
	$GREEDY_{20\%}$	**99.55**	99.31	**0.00**	18.6	5.31	29
DT	$1R_{min\_cl=3}$	80.04	74.09	9.14	2976.2	1.00	241
	I-REP	95.59	**98.92**	10.47	**6.0**	**2.85**	**5**
	RIPPER0	98.40	98.12	1.09	15.6	4.50	21
	RIPPER1	98.58	98.53	1.34	15.0	4.32	66
	RIPPER2	98.67	98.72	1.44	15.4	4.13	102
	$GREEDY_{20\%}$	**98.87**	98.25	**0.00**	23.4	6.03	43
kNN	$1R_{min\_cl=3}$	81.67	76.65	9.30	2257.8	1.00	240
	I-REP	95.09	**99.02**	11.99	**6.6**	**2.97**	**5**
	RIPPER0	98.45	98.22	1.14	15.2	4.54	21
	RIPPER1	98.67	98.64	1.27	16.2	4.40	67
	RIPPER2	98.83	98.86	1.24	20.2	3.99	103
	$GREEDY_{20\%}$	**98.84**	98.19	**0.00**	22.4	6.08	43
AdaBoost	$1R_{min\_cl=3}$	81.47	72.04	1.23	2911.8	1.00	255
	I-REP	95.33	**99.04**	11.49	**5.8**	**3.02**	**5**
	RIPPER0	98.54	98.28	0.99	13.6	4.25	21
	RIPPER1	98.79	98.68	1.01	14.8	4.19	64
	RIPPER2	98.87	98.75	0.91	15.6	4.21	107
	$GREEDY_{20\%}$	**98.93**	98.34	**0.00**	19.8	5.98	38

Similar experiments were performed with the EMBER dataset. Since EMBER comes with its own test set, we have trained the ML algorithms using the train set only. The results can be seen in Table 4.1. The training of the RBCs was done using the ML algorithms' predictions on the test set. Obtained results are shown in Table 4.3.

We can determine that previous outcomes on Kozák's dataset apply to the EMBER dataset, too. However, there is a significant drop in both accuracy and sensitivity. We could argue that this drop is caused by the ML algorithms not performing too well with our current settings. On the other hand, the worst performing algorithm LR was described almost perfectly by the RBCs. Observe that for LR, the decision list sizes and the sizes of their rules are quite small compared to other ML algorithms. Even though minimising the TDL metric (see Definition 12) should prevent prematurely ending rule learning, we can see that this may not always be the case, as with kNN. It may be

Table 4.3: Interpreting ML algorithms' results on the EMBER dataset using RBCs

time (s) shows the approximate time in seconds for training of RBCs

	RBC	ACC	TPR	FPR	DL size	ø r size	time (s)
LR	I-REP	97.14	**99.91**	15.74	**19**	**2.32**	**202**
	RIPPER0	99.77	99.76	0.20	24	3.92	234
	RIPPER1	99.80	99.80	0.20	27	3.78	807
	RIPPER2	**99.82**	99.82	0.22	28	3.71	1445
	$\text{GREEDY}_{20\%}$	99.77	99.72	**0.00**	33	4.67	662
RF	I-REP	91.94	**98.95**	15.90	**53**	**7.40**	**659**
	RIPPER0	97.83	96.36	0.51	98	9.80	2937
	RIPPER1	98.30	97.45	0.75	105	9.65	10792
	RIPPER2	**98.43**	97.70	0.75	112	9.73	18726
	$\text{GREEDY}_{20\%}$	98.31	96.80	**0.00**	151	10.94	5039
DT	I-REP	86.82	**92.80**	19.48	**35**	**8.83**	**1322**
	RIPPER0	89.37	80.53	1.30	127	11.68	8901
	RIPPER1	91.03	85.23	2.85	128	11.03	41599
	RIPPER2	**91.71**	87.72	4.07	122	10.44	67220
	$\text{GREEDY}_{20\%}$	89.02	78.62	**0.00**	211	11.90	10106
kNN	I-REP	79.46	**86.05**	26.90	**37**	**6.97**	**1223**
	RIPPER0	77.38	54.25	0.29	57	9.75	4291
	RIPPER1	79.38	58.42	0.40	75	9.44	14938
	RIPPER2	**79.75**	59.19	0.40	78	9.45	31445
	$\text{GREEDY}_{20\%}$	75.86	50.84	**0.00**	64	11.23	3096
AdaBoost	I-REP	87.58	**96.39**	22.47	**30**	**7.97**	**960**
	RIPPER0	94.02	89.85	1.23	138	10.93	8488
	RIPPER1	94.64	91.93	2.27	133	10.45	39098
	RIPPER2	**95.08**	93.23	2.82	140	10.21	64606
	$\text{GREEDY}_{20\%}$	92.09	85.15	**0.00**	181	11.75	9762

necessary to increase the largest bit difference (see Algorithm 6).

4.4.1 Pruning and Metrics

In Section 1.4, we have covered how both I-REP and RIPPER handle noisy data – by utilising pruning. Cohen [19] pointed out that I-REP's incapability of converging towards better solutions is mainly caused by its pruning metric. We were interested in knowing how much the pruning metric affects learning and whether the cause of RIPPER's FPR increase during optimisation phases lies in the usage of Metric (1.14).

Since the multiclass problem can be reduced to an alternating two-class problem (see Section 1.4.4), we can understand pruning metrics as two-variable functions. Fortunately, this number is perfect for a better understanding pruning metrics by graphing them. Figure 4.1 shows some of the pruning metrics we have used in our experiments. We have simplified I-REP's pruning metric as it can be viewed as a plane for fixed P and N (see Metric (1.14)). Here lies the key reason why I-REP tends to make bad decisions when pruning; points with a different number of malicious and benign samples are often indistin-

Table 4.4: Testing different metrics on Kozák's dataset using RIPPER0

	RBC	ACC	TPR	FPR	DL size	ø r size
LR	RIPPER0	98.83	98.57	**0.70**	12.4	3.87
	RIPPER0$_{\sqrt{each}}$	**98.85**	**99.10**	1.60	11.6	3.74
	RIPPER0$_{\sqrt{both}}$	98.70	99.00	1.83	10.4	3.47
	RIPPER0$_{p^{-1}}$	98.47	99.00	2.47	**8.8**	**3.22**
RF	RIPPER0	**99.25**	99.06	**0.38**	14.2	4.11
	RIPPER0$_{\sqrt{each}}$	99.23	99.17	0.65	14.2	3.75
	RIPPER0$_{\sqrt{both}}$	98.71	**99.54**	2.81	14.2	2.90
	RIPPER0$_{p^{-1}}$	98.44	98.20	1.10	**10.0**	**3.40**
DT	RIPPER0	**98.40**	98.12	**1.09**	15.6	4.50
	RIPPER0$_{\sqrt{each}}$	98.10	98.60	2.81	12.0	3.73
	RIPPER0$_{\sqrt{both}}$	97.79	**98.78**	4.00	**8.8**	**3.32**
	RIPPER0$_{p^{-1}}$	97.59	98.60	4.26	8.6	3.59
kNN	RIPPER0	**98.45**	98.22	**1.14**	15.2	4.54
	RIPPER0$_{\sqrt{each}}$	98.21	98.79	2.82	12.8	3.80
	RIPPER0$_{\sqrt{both}}$	97.91	98.76	3.62	9.6	3.36
	RIPPER0$_{p^{-1}}$	97.09	**99.05**	6.45	**7.8**	**3.26**
AdaBoost	RIPPER0	**98.54**	98.28	**0.99**	13.6	4.25
	RIPPER0$_{\sqrt{each}}$	98.13	98.51	2.56	9.8	3.65
	RIPPER0$_{\sqrt{both}}$	97.98	**98.98**	3.84	9.6	3.23
	RIPPER0$_{p^{-1}}$	97.19	97.58	3.51	**7.8**	**3.31**

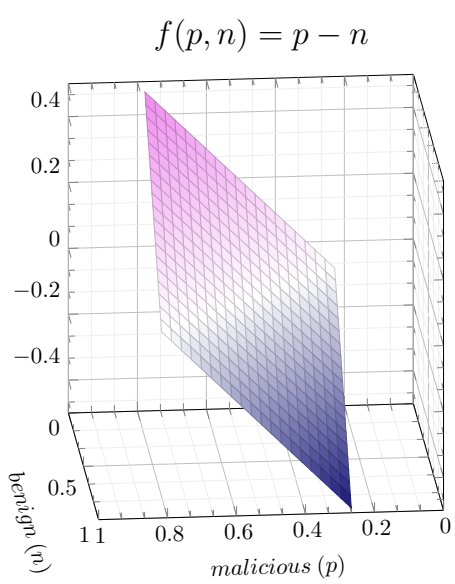
guishable. RIPPER's pruning metric seems to have good characteristics; the only issue we could identify is that it does not differentiate between positive samples when no negative samples are present.

We tried several different metrics and picked three that performed well on Kozák's dataset:

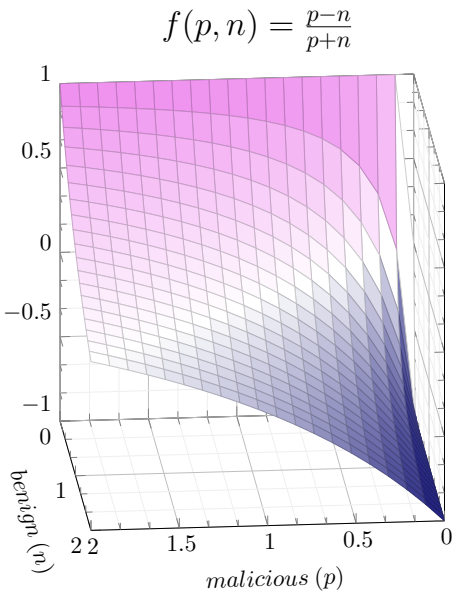
$$\underbrace{\frac{p-n}{\sqrt{p}+\sqrt{n}}}_{\sqrt{each}}, \quad \underbrace{\frac{p-n}{\sqrt{p+n}}}_{\sqrt{both}}, \quad \underbrace{\frac{p-np^{-1}}{\sqrt{p}+\sqrt{n}}}_{p^{-1}}, \tag{4.7}$$

where p is the number of positive samples and n is the number of negative samples in the pruning set. While designing the metrics, we searched for functions with similar properties as Metric (1.17). Metrics were tested using both I-REP and RIPPER0. However, since the results did not differ significantly, we only list those used with RIPPER0 in Table 4.4. Metrics $\sqrt{each}$ and $\sqrt{both}$ seem to outperform RIPPER's pruning metric sensitivity wise. On the other hand, the fall-out rate increases several times, too. There seems to

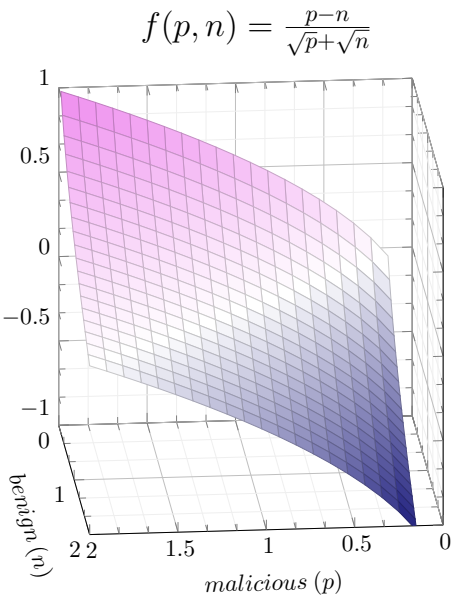
(a) Simplified IREP's pruning metric



(b) RIPPER's pruning metric



(c) Scaled RIPPER's pruning metric



(d) Scaled function with a saddle point

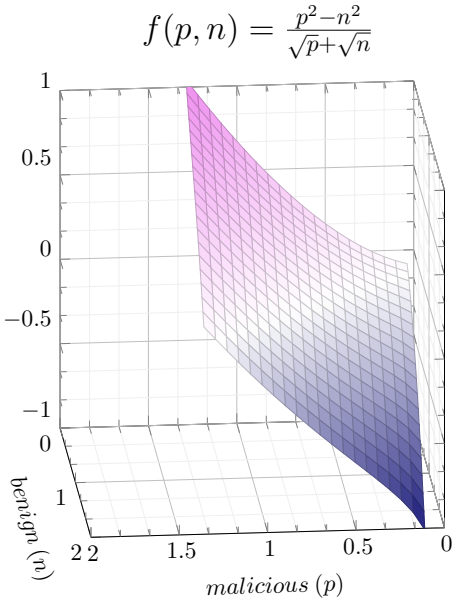


Figure 4.1: Understanding pruning metrics as 3D graphs
Pruning metrics should have the following properties: If only malicious files are present, pruning metrics should be at their maximum. For benign files only, they should be at their minimum. Otherwise, they need to compromise, and should take into account a lower number of benign files.

Table 4.5: Testing different metrics on the EMBER dataset using RIPPER0

	RBC	ACC	TPR	FPR	DL size	ø r size
LR	RIPPER0	99.77	99.76	0.20	24	3.92
	RIPPER0$_{\sqrt{each}}$	99.79	99.78	**0.16**	**23**	**3.17**
	RIPPER0$_{\sqrt{both}}$	99.79	**99.83**	0.39	25	3.40
	RIPPER0$_{p^{-1}}$	**99.80**	**99.83**	0.31	25	3.28
RF	RIPPER0	**97.83**	96.36	**0.51**	98	9.80
	RIPPER0$_{\sqrt{each}}$	97.33	97.76	3.14	69	8.62
	RIPPER0$_{\sqrt{both}}$	96.73	**98.26**	4.99	67	8.49
	RIPPER0$_{p^{-1}}$	95.03	96.42	6.54	**17**	**7.18**
DT	RIPPER0	**89.37**	80.53	**1.30**	127	11.68
	RIPPER0$_{\sqrt{each}}$	89.27	89.99	11.49	58	9.93
	RIPPER0$_{\sqrt{both}}$	88.22	**90.59**	14.29	50	9.58
	RIPPER0$_{p^{-1}}$	77.79	63.37	6.99	**6**	**8.00**
kNN	RIPPER0	77.38	54.25	0.29	57	9.75
	RIPPER0$_{\sqrt{each}}$	75.69	52.48	1.91	12	7.75
	RIPPER0$_{\sqrt{both}}$	**83.11**	**73.80**	7.91	62	8.95
	RIPPER0$_{p^{-1}}$	64.52	27.77	**0.00**	**3**	**7.33**
AdaBoost	RIPPER0	**94.02**	89.85	**1.23**	138	10.93
	RIPPER0$_{\sqrt{each}}$	93.78	93.63	6.06	81	9.70
	RIPPER0$_{\sqrt{both}}$	92.77	**93.65**	8.23	75	9.37
	RIPPER0$_{p^{-1}}$	88.44	89.65	12.93	**11**	**7.36**

be a trend of more compact decision lists for our designed metrics. This could be a potential trade-off for the otherwise worse performance.

Using the EMBER dataset, we tried to verify our results. Our results show that metrics $\sqrt{each}$ and $\sqrt{both}$ aim for higher TPR; however, as we saw for Kozák's dataset, they often increase the fall-out rate several times. Unlike Kozák's dataset, the drop in the total number of conditions is more significant. kNN seems to be overall a special case. All metrics, except $\sqrt{both}$, are not performing good. Notice that metric p^{-1} is in some cases outperformed by I-REP, as seen in Table 4.3.

We did not record any improvement nor deterioration upon exchanging Metric (1.14) during RIPPER's optimisation phase.

4.4.2 Removing Rules in RIPPER

We tested two additional rule removal approaches mentioned in Section 3.4 – RIPPER$k_{\text{remove_all}}$ and RIPPERk_{sort}. In this section, we highlight columns

with better performance than RIPPERk. We were expecting the total number of conditions to drop. Our results on Kozák's dataset in Table 4.6 confirm this, although the drop is not very significant. Both methods seem to perform worse otherwise; however, after closer inspecting values in Table 4.2, we can see that these differences are minimal.

We ran a similar experiment for the EMBER dataset and ended up with similar results, as shown in Table 4.7. This experiment also shows that removing all rules (RIPPER$k_{\text{remove_all}}$) may be too greedy; it can lead to higher FPR and a lower TPR, as seen for DecisionTree.

Table 4.6: Using different rule removal approaches (remove_all and sort) in RIPPERk for Kozák's dataset

Columns where rule removal approaches performed better than RIPPERk in Table 4.2 are highlighted.

	RBC	ACC	TPR	FPR	DL size	ø r size
LR	RIPPER0$_{\text{remove_all}}$	96.86	95.53	0.82	**10.8**	**3.75**
	RIPPER0$_{\text{sort}}$	98.77	**98.58**	0.89	**11.4**	**3.78**
	RIPPER1$_{\text{remove_all}}$	98.25	98.11	1.51	**13.6**	**2.91**
	RIPPER1$_{\text{sort}}$	99.06	**99.08**	0.96	**13.0**	**3.71**
	RIPPER2$_{\text{remove_all}}$	98.65	99.10	2.16	**17.2**	**2.84**
	RIPPER2$_{\text{sort}}$	99.17	99.25	0.97	**14.6**	**3.66**
RF	RIPPER0$_{\text{remove_all}}$	98.37	97.68	**0.37**	**12.2**	**4.05**
	RIPPER0$_{\text{sort}}$	99.14	98.88	0.39	**13.4**	**4.12**
	RIPPER1$_{\text{remove_all}}$	99.29	**99.36**	0.83	**19.2**	**3.24**
	RIPPER1$_{\text{sort}}$	99.37	99.28	0.45	**14.6**	**3.96**
	RIPPER2$_{\text{remove_all}}$	99.50	**99.66**	0.78	25.4	2.88
	RIPPER2$_{\text{sort}}$	99.41	99.31	0.42	**14.8**	**3.91**
DT	RIPPER0$_{\text{remove_all}}$	98.09	97.64	1.11	**13.8**	**4.30**
	RIPPER0$_{\text{sort}}$	98.37	**98.15**	1.22	**15.2**	**4.30**
	RIPPER1$_{\text{remove_all}}$	98.42	98.35	1.44	**14.2**	**4.11**
	RIPPER1$_{\text{sort}}$	**98.70**	**98.78**	1.46	17.0	4.15
	RIPPER2$_{\text{remove_all}}$	98.39	98.79	2.34	**16.2**	**3.50**
	RIPPER2$_{\text{sort}}$	**98.71**	**98.79**	**1.43**	16.4	4.12
kNN	RIPPER0$_{\text{remove_all}}$	97.52	96.79	1.15	**14.6**	**4.34**
	RIPPER0$_{\text{sort}}$	**98.51**	**98.35**	1.20	15.8	4.43
	RIPPER1$_{\text{remove_all}}$	98.04	97.75	1.43	**16.8**	**3.59**
	RIPPER1$_{\text{sort}}$	98.66	98.63	1.28	**16.0**	**4.21**
	RIPPER2$_{\text{remove_all}}$	98.69	**98.93**	1.73	**20.6**	**3.46**
	RIPPER2$_{\text{sort}}$	98.78	98.76	1.19	**16.8**	**4.22**
AdaBoost	RIPPER0$_{\text{remove_all}}$	98.52	98.22	0.93	14.0	4.25
	RIPPER0$_{\text{sort}}$	**98.55**	98.27	**0.93**	14.2	4.31
	RIPPER1$_{\text{remove_all}}$	94.84	92.54	**0.94**	**11.2**	**3.94**
	RIPPER1$_{\text{sort}}$	98.75	98.65	1.07	**14.4**	**4.10**
	RIPPER2$_{\text{remove_all}}$	98.10	98.67	2.94	**15.8**	**3.43**
	RIPPER2$_{\text{sort}}$	**98.89**	**98.85**	1.03	**15.4**	**4.06**

Table 4.7: Using different rule removal approaches (remove_all and sort) in RIPPERk for the EMBER dataset

Columns where rule removal approaches performed better than RIPPERk in Table 4.3 are highlighted.

	RBC	ACC	TPR	FPR	DL size	ø r size
LR	$\text{RIPPER0}_{\text{remove\_all}}$	99.74	99.71	**0.15**	**22**	**3.91**
	$\text{RIPPER0}_{\text{sort}}$	99.74	99.71	**0.15**	**22**	**3.91**
	$\text{RIPPER1}_{\text{remove\_all}}$	**99.81**	99.80	**0.12**	**25**	**3.72**
	$\text{RIPPER1}_{\text{sort}}$	**99.81**	99.80	**0.14**	**25**	**3.68**
	$\text{RIPPER2}_{\text{remove\_all}}$	**99.83**	99.82	**0.13**	**27**	**3.67**
	$\text{RIPPER2}_{\text{sort}}$	**99.84**	**99.85**	**0.18**	**28**	**3.61**
RF	$\text{RIPPER0}_{\text{remove\_all}}$	**98.11**	**96.95**	0.60	106	9.96
	$\text{RIPPER0}_{\text{sort}}$	**98.13**	**97.01**	0.60	108	9.98
	$\text{RIPPER1}_{\text{remove\_all}}$	**98.39**	**97.59**	**0.71**	112	9.88
	$\text{RIPPER1}_{\text{sort}}$	**98.45**	**97.78**	0.81	119	9.78
	$\text{RIPPER2}_{\text{remove\_all}}$	**98.44**	**97.88**	0.93	**110**	**9.67**
DT	$\text{RIPPER0}_{\text{remove\_all}}$	86.82	75.04	**0.76**	**93**	**11.90**
	$\text{RIPPER0}_{\text{sort}}$	86.82	75.04	**0.76**	**93**	**11.90**
	$\text{RIPPER1}_{\text{remove\_all}}$	89.94	83.26	3.01	**100**	**10.86**
	$\text{RIPPER1}_{\text{sort}}$	90.54	84.49	3.08	**115**	**11.09**
	$\text{RIPPER2}_{\text{remove\_all}}$	82.01	69.72	5.02	**99**	**10.12**
	$\text{RIPPER2}_{\text{sort}}$	91.32	87.27	4.40	**97**	**10.33**
kNN	$\text{RIPPER0}_{\text{remove\_all}}$	76.14	51.56	**0.14**	**44**	**9.86**
	$\text{RIPPER0}_{\text{sort}}$	76.14	51.56	**0.14**	**44**	**9.86**
	$\text{RIPPER1}_{\text{remove\_all}}$	79.16	58.09	0.49	**70**	**9.61**
	$\text{RIPPER1}_{\text{sort}}$	**79.47**	**58.73**	0.51	76	9.67
	$\text{RIPPER2}_{\text{remove\_all}}$	**80.19**	**60.33**	0.64	77	9.73
	$\text{RIPPER2}_{\text{sort}}$	**79.93**	**59.79**	0.63	83	9.76
AdaBoost	$\text{RIPPER0}_{\text{remove\_all}}$	93.27	88.31	**1.07**	**120**	**11.05**
	$\text{RIPPER0}_{\text{sort}}$	93.46	88.71	**1.12**	**126**	**11.10**
	$\text{RIPPER1}_{\text{remove\_all}}$	93.73	90.17	**2.21**	**107**	**10.26**
	$\text{RIPPER1}_{\text{sort}}$	**94.83**	**92.43**	2.44	138	10.51
	$\text{RIPPER2}_{\text{remove\_all}}$	94.13	91.74	3.16	**97**	**9.77**
	$\text{RIPPER2}_{\text{sort}}$	95.04	**93.24**	2.90	**135**	**10.22**

Chapter 5

Discussion

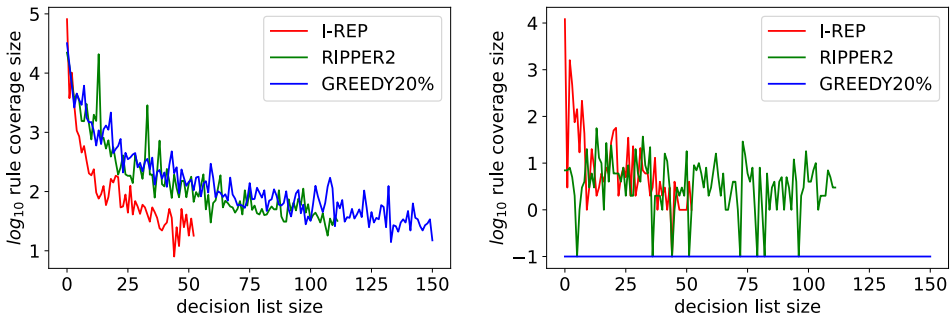
This chapter discusses the results obtained in Chapter 4. We further discuss different behaviours of RBCs and bring another view for a better comparison. Lastly, we compare our approach with other researchers covering a similar topic.

5.1 Experiments Review

Our experiments show that RBCs could be a good tool to interpret the results of ML algorithms. Out of the four algorithms we used, RIPPERk performed the best. It was able to keep a good accuracy and still more compact decision lists than GREEDY$_{20\%}$. RIPPER0's outcomes could be more useful for malware detection as it often had the lowest FPR (in RIPPER's iterations). The FPR increase may be caused by rule removal (see Algorithm 5), although different removal techniques do not necessarily have much of an impact (see Section 4.4.2). RIPPER2$_{\text{sort}}$ required too much computational time for the output of RF. Thus it is not included in Table 4.7.

The simplest algorithm, 1R, did not perform too well. However, it could identify good features for other ML algorithms. In our case, we used it to remove features that did not seem to have any significance as to whether the file is malicious or not (see Section 4.2.3). We did not show results for 1R for the EMBER dataset since the Python implementation is not efficient enough.

We further analysed the behaviours of RBCs. We were interested in knowing how much the RBCs' performance differs over time. Figure 5.1a shows the rule coverage size for malicious samples for the EMBER dataset. Figure 5.1b shows this for benign files. Both figures use RandomForest's outcome; the rest of the figures can be found in Figure B.1 in Appendix. We see that I-REP starts by covering many more samples than the other two algorithms, both malicious and benign. Our algorithm GREEDY$_{20\%}$ does not cover any benign samples and seems to be following a similar path as RIPPER2 does.



(a) True positives (b) False positives

Figure 5.1: Rule coverage size – RandomForest & EMBER
Both graphs show how rules cover different samples over time. The y-axis is log-scaled and represents covered samples; the x-axis represents a decision list size. Both graphs use RandomForest's output on the EMBER dataset. Value −1 on the y-axis corresponds to no covered samples.

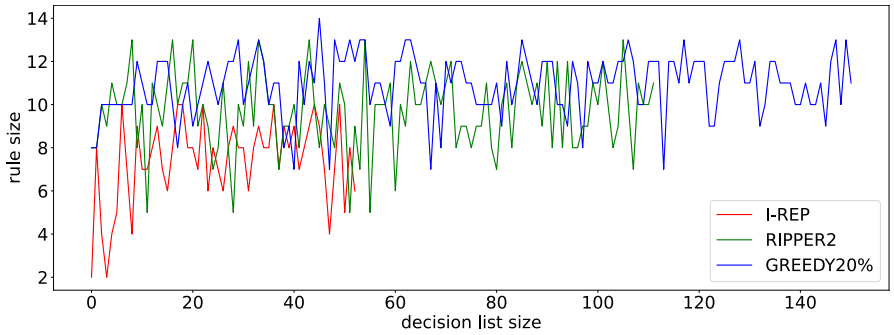


Figure 5.2: Visualisation of rule sizes over time – RandomForest & EMBER
The x-axis corresponds to rule sizes – a number of conditions; the y-axis corresponds to decision list size. For example, the 0th rule learned by I-REP has a size of two, the 20th rule learned has a size of eight.

Not covering any benign samples probably means that the stop condition we chose does not have much of an impact. On the other hand, this is subject to further testing with more datasets. We can see that RIPPER2 has a few spikes along the way. Since both graphs are log-scaled, these spikes mean RIPPER2 has been able to find rules with great coverage size. This brings up an interesting question for further study: could we potentially shift those spikes to have them occur as soon as possible? Perhaps by changing the order of the rules?

Figure 5.2 displays differences in rule sizes over time. Again, we used RandomForest's outcome for the EMBER dataset; the rest can be found in Figure B.2 in Appendix. We would expect I-REP to have a small number of conditions. However, it is surprising that this number gradually increases

with more rules. Both RIPPER2 and $\text{GREEDY}_{20\%}$ are more stable in this case. We cannot directly compare each rule size as every rule is dependent on the previous one.

5.2 Comparison with Related Works

Feng et al. [15] combined three different approaches for malware detection – hash-based approach, SVM-based approach and rule-based approach. Every approach is intended to be used for malware classes (e.g. Trojan, Spyware) with different distributions. Instead of building features from PE file headers (see Chapter 2), they used n-grams based on the content of a binary file – n-grams are all subparts of a string, with length n [59]. In their experiments, they reduced space complexity from 1.8MB (signature-based approach) to 17.9KB (combined approach).

Schultz et al. [60] compared RIPPER with other ML algorithms in malware detection on previously unseen samples. The paper does not clarify the number of iterations used for RIPPER. They used static features extracted from the PE files – used DLLs, DLL function calls and count of DLL function calls. They discussed how malware developers could use the information generated by the classifiers to modify their malware, for example, by changing resource usage.

Our work focuses on interpreting the results obtained by various ML algorithms. Unlike the aforementioned works, our approach did not use any train data; instead, we used the outcomes of ML algorithms to train the RBCs. We cannot determine the accuracy of our approach on unseen data; this is subject to further research. To offer a less biased view, we have used two publicly available datasets. Thus our results should be easily verifiable. We also tried to modify the behaviours of some of the algorithms by changing some of the steps.

Conclusion

The main focus of this thesis was to verify whether rule-based classifiers would be useful in interpreting the results of other machine learning algorithms. This chapter summarises our approach and contribution to this topic and discusses future work.

Contribution

Chapter 1 laid necessary theoretical knowledge for rule-based classifiers. We reviewed several papers and picked four rule learning algorithms. We discussed some parts of the algorithms that were not clarified in the papers and suggested an approach to solving them. We implemented three of the algorithms mentioned in Chapter 1 and an algorithm with a different approach to rule learning. Chapter 3 gave details about the implementations.

In Chapter 2 we briefly described the inner structure of PE files. It helped us better understand the datasets we were working with. We thoroughly analysed Kozák's dataset, transformed it and selected potentially good features for classification. The EMBER [16] dataset served mainly for comparison, and we used available functions from the authors to transform it. This process is depicted in Chapter 4.

Chapter 4 analysed the performance of rule-based classifiers on the outcomes of machine learning algorithms and further examined the behaviours of some of the algorithms – mainly RIPPERk's. Our results showed that rule-based classifiers have great potential to interpret the outcomes of machine learning algorithms – again, mainly RIPPER. GREEDY$_{20\%}$, which we proposed in this work, may serve as an alternative tool if the error should be as low as possible. We have also shown that I-REP and RIPPER algorithm behaviours can be dramatically affected by using different pruning metrics. RIPPER's behaviour is most likely not affected by using a different rule removal approach.

Rule-based classifiers could reduce the space needed to store the results of machine learning algorithms as well as bring some light upon possible decisions of machine learning algorithms.

Future Work

Although we have created our implementations of rule-based classifiers, there is much room for improvement. This would mainly require creating a better underlying structure in a more appropriate language, such as C++. One could then connect the parts implemented in C++ and Python by using, e.g. Cython – a tool for writing C/C++ extensions for Python [61]. Such an implementation could be beneficial for other researchers as well.

Algorithm $\text{GREEDY}_{20\%}$ could serve as a relatively good tool for precise results interpretation. However, it should be verified that its performance is better than, e.g. RIPPER's performance without pruning. Other stop conditions should be built, too.

The methods we used would be useful for offline learning. On the other hand, if a decision list is grown and new data is obtained, simply regrowing the decision list on all data would not be as efficient. Thus, other approaches are needed. We think that RIPPER's optimisation phase could offer a solution; however, it is still necessary to explain its worsening behaviour in some cases.

Bibliography

1. SZOR, Peter. *Art of Computer Virus Research and Defense*. Pearson Education, 2005. ISBN 0321304543.
2. KŘOUSTEK, Jakub. *WannaCry ransomware that infected Telefonica and NHS hospitals is spreading aggressively, with over 50,000 attacks so far today* [online] [visited on 2021-04-28]. Available from: `https://blog.avast.com/ransomware-that-infected-telefonica-and-nhs-hospitals-is-spreading-aggressively-with-over-50000-attacks-so-far-today`.
3. SEGUIN, Patrick. *The Essential Guide to Ransomware* [online] [visited on 2021-04-28]. Available from: `https://www.avast.com/c-what-is-ransomware`.
4. KREBS, Brian. *Who is Anna-Senpai, the Mirai Worm Author?* [Online] [visited on 2021-04-28]. Available from: `https://krebsonsecurity.com/2017/01/who-is-anna-senpai-the-mirai-worm-author/`.
5. EGELE, Manuel; SCHOLTE, Theodoor; KIRDA, Engin; KRUEGEL, Christopher. A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*. 2008, vol. 44, no. 2, pp. 1–42. Available from DOI: `10.1145/2089125.2089126`.
6. WILLEMS, Carsten; HOLZ, Thorsten; FREILING, Felix. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*. 2007, vol. 5, no. 2, pp. 32–39. Available from DOI: `10.1109/MSP.2007.45`.
7. ELDER, Jeff. *Is machine learning useful for cybersecurity?* [Online] [visited on 2021-04-29]. Available from: `https://blog.avast.com/avast-explains-cybersecurity-ai-at-enigma-conference`.
8. GUPTA, Rajarshi. *Why is Security a unique challenge for AI* [online] [visited on 2021-04-29]. Available from: `https://blog.avast.com/why-security-is-a-unique-challenge-for-ai`.

9. MOLNAR, Christoph. *Interpretable Machine Learning - Decision Rules* [online] [visited on 2021-02-18]. Available from: https://christophm.github.io/interpretable-ml-book/rules.html.

10. DOSHI-VELEZ, Finale; KIM, Been. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*. 2017.

11. GILPIN, Leilani H.; BAU, David; YUAN, Ben Z.; BAJWA, Ayesha; SPECTER, Michael; KAGAL, Lalana. Explaining explanations: An overview of interpretability of machine learning. In: *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*. 2018, pp. 80–89. Available from DOI: 10.1109/DSAA.2018.00018.

12. ANGWIN, Julia; LARSON, Jeff; MATTU, Surya; KIRCHNER, Lauren. *Machine Bias* [online]. 2016 [visited on 2021-04-29]. Available from: https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing.

13. WITTEN, Ian H.; FRANK, Eibe; HALL, Mark A. *Data Mining: Practical Machine Learning Tools and Techniques*. 2011. ISBN 978-0-12-374856-0.

14. PANDA, B. S. *Rule Based Classification* [online] [visited on 2020-11-04]. Available from: https://web.iitd.ac.in/~bspanda/rb.pdf.

15. FENG, Zhentan; AL., et. HRS: A Hybrid Framework for Malware Detection. In: *Proceedings of the 2015 ACM International Workshop on International Workshop on Security and Privacy Analytics* [online]. New York, NY, USA: Association for Computing Machinery, 2015, pp. 19–26 [visited on 2021-01-27]. ISBN 9781450333412. Available from DOI: 10.1145/2713579.2713585.

16. ANDERSON, Hyrum S.; ROTH, Phil. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*. 2018. Available from arXiv: 1804.04637 [cs.CR].

17. QUINLAN, John Ross. *C4.5: Programs for Machine Learning*. 1993. ISBN 978-1-55860-238-0.

18. CENDROWSKA, Jadzia. PRISM: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies*. 1987, vol. 27, no. 4, pp. 349–370. Available from DOI: 10.1016/S0020-7373(87)80003-2.

19. COHEN, William W. Fast effective rule induction. In: *Machine learning proceedings 1995*. Elsevier, 1995, pp. 115–123. Available from DOI: 10.1016/B978-1-55860-377-6.50023-2.

20. HOLTE, Robert C. Very simple classification rules perform well on most commonly used datasets. *Machine learning*. 1993, vol. 11, no. 1, pp. 63–90. Available from DOI: 10.1023/A:1022631118932.

21. MITCHELL, Tom M. *Machine Learning*. 1997. ISBN 0070428077.

22. CLARK, Peter; NIBLETT, Tim. The CN2 Induction Algorithm. *Machine learning*. 1989, vol. 3, no. 4, pp. 261–283. Available from DOI: 10.1023/A:1022641700528.

23. QUINLAN, John Ross. Learning efficient classification procedures and their application to chess end games. In: *Machine learning*. Springer, 1983, pp. 463–482.

24. MICHALSKI, Ryszard S. On the quasi-minimal solution of the general covering problem. 1969.

25. KALBFLEISCH, James G. *Probability and Statistical Inference II*. 1979. ISBN 978-1-4684-0091-5.

26. FÜRNKRANZ, Johannes; WIDMER, Gerhard. Incremental reduced error pruning. In: *Machine Learning Proceedings 1994*. Elsevier, 1994, pp. 70–77. ISBN 978-1-55860-335-6. Available from DOI: 10.1016/B978-1-55860-335-6.50017-9.

27. QUINLAN, J Ross; CAMERON-JONES, R Mike. FOIL: A midterm report. In: *European conference on machine learning*. 1993, pp. 1–20. ISBN 978-3-540-56602-1.

28. DAIN, Oliver; CUNNINGHAM, Robert K.; BOYER, Stephen. Irep++, a faster rule learning algorithm. In: *Proceedings of the 2004 SIAM International Conference on Data Mining*. 2004, pp. 138–146. Available from DOI: 10.1137/1.9781611972740.13.

29. FIDELIS, Marcos Vinicius; LOPES, Heitor S.; FREITAS, Alex A. Discovering comprehensible classification rules with a genetic algorithm. In: *Proceedings of the 2000 congress on evolutionary computation. cec00 (cat. no. 00th8512)*. 2000, vol. 1, pp. 805–810. Available from DOI: 10.1109/CEC.2000.870381.

30. WHITLEY, Darrell. A genetic algorithm tutorial. *Statistics and computing*. 1994, vol. 4, no. 2, pp. 65–85. Available from DOI: 10.1007/BF00175354.

31. CRAVEN, Mark; SHAVLIK, Jude. Extracting tree-structured representations of trained networks. *Advances in neural information processing systems*. 1995, vol. 8, pp. 24–30. Available from DOI: 10.5555/2998828.2998832.

32. MILLER, Michael. *A Brief History of Microsoft Windows* [online] [visited on 2021-04-11]. Available from: https://www.informit.com/articles/article.aspx?p=1358665&seqNum=4.

33. PIETREK, Matt. *An In-Depth Look into the Win32 Portable Executable File Format* [online] [visited on 2021-04-06]. Available from: https://docs.microsoft.com/en-us/archive/msdn-magazine/2002/february/inside-windows-win32-portable-executable-file-format-in-detail.

34. OS DEV CONTRIBUTORS. *PE* [online] [visited on 2021-04-06]. Available from: https://wiki.osdev.org/PE.

35. MICROSOFT DOCUMENTATION CONTRIBUTORS. */STUB (MS-DOS Stub File Name)* [online] [visited on 2021-04-06]. Available from: https://docs.microsoft.com/en-us/cpp/build/reference/stub-ms-dos-stub-file-name.

36. DABAK, Prasad; PHADKE, Sandeep; BORATE, Milind. *Undocumented Windows NT*. John Wiley & Sons, Inc., 1999. ISBN 0764545698.

37. CARRERA, Ero. *PE Header Walkthrough* [online] [visited on 2021-04-06]. Available from: https://drive.google.com/file/d/0B3_wGJkuWLytQmc2di0wajB1Xzg/view.

38. KOWALCZYK, Krzysztof. *Portable Executable File Format* [online] [visited on 2021-04-06]. Available from: https://blog.kowalczyk.info/articles/pefileformat.html.

39. VIRTANEN, Pauli; AL., et. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*. 2020, vol. 17, pp. 261–272. Available from DOI: 10.1038/s41592-019-0686-2.

40. DEVELOPERS, NumPy. *What is NumPy?* [Online]. 2021 [visited on 2021-05-01]. Available from: https://numpy.org/doc/stable/user/whatisnumpy.html.

41. HARRIS, Charles R.; AL., et. Array programming with NumPy. *Nature*. 2020, vol. 585, pp. 357–362. Available from DOI: 10.1038/s41586-020-2649-2.

42. DEVELOPERS, scikit-learn. *How to optimize for speed* [online] [visited on 2021-04-14]. Available from: https://scikit-learn.org/stable/developers/performance.html.

43. KLUYVER, Thomas; AL., et. Jupyter Notebooks – a publishing format for reproducible computational workflows. In: LOIZIDES, Fernando; SCMIDT, Birgit (eds.). *Positioning and Power in Academic Publishing: Players, Agents and Agendas* [online]. IOS Press, 2016, pp. 87–90 [visited on 2021-05-02]. ISBN 978-1-61499-649-1. Available from: https://eprints.soton.ac.uk/403913/.

44. CPYTHON DEVELOPERS. *pickle — Python object serialization* [online] [visited on 2021-05-12]. Available from: https://docs.python.org/3/library/pickle.html.

45. KOZÁK, Matouš. *Malware detection dataset* [online] [visited on 2021-05-12]. Available from: https://github.com/matouskozak/malware_detection_dataset.

46. PEDREGOSA, F.; AL., et. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011, vol. 12, pp. 2825–2830.

47. KOZÁK, Matouš. *Static malware detection using recurrent neural networks*. Prague, 2020. Bachelor's Thesis. Faculty of Information Technology, Czech Technical University.

48. RAJARAMAN, Anand; ULLMAN, Jeffrey. *Mining of Massive Datasets*. Cambridge University Press, 2011. ISBN 9781139924801.

49. DEVELOPERS, scikit-learn. *sklearn.feature_extraction.text.TfidfVectorizer* [online] [visited on 2021-05-02]. Available from: `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html`.

50. DEVELOPERS, scikit-learn. *sklearn.feature_extraction.FeatureHasher* [online] [visited on 2021-04-20]. Available from: `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.FeatureHasher.html`.

51. ANDERSON, Hyrum S.; ROTH, Phil. *Elastic Malware Benchmark for Empowering Researchers* [online] [visited on 2021-05-12]. Available from: `https://github.com/elastic/ember`.

52. DEVELOPERS, scikit-learn. *6.3. Preprocessing data* [online] [visited on 2021-04-20]. Available from: `https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler`.

53. DEVELOPERS, scikit-learn. *1.13. Feature selection* [online] [visited on 2021-04-20]. Available from: `https://scikit-learn.org/stable/modules/feature_selection.html#select-from-model`.

54. HOARE, Jake. *How is Variable Importance Calculated for a Random Forest?* [Online] [visited on 2021-04-20]. Available from: `https://www.displayr.com/how-is-variable-importance-calculated-for-a-random-forest/`.

55. AZNAR, Pablo. *What is the difference between Extra Trees and Random Forest?* [Online] [visited on 2021-04-20]. Available from: `https://quantdare.com/what-is-the-difference-between-extra-trees-and-random-forest/`.

56. ZHAO, Kai. *Feature Extraction using Principal Component Analysis – A Simplified Visual Demo* [online] [visited on 2021-05-13]. Available from: `https://towardsdatascience.com/feature-extraction-using-principal-component-analysis-a-simplified-visual-demo-e5592ced100a`.

57. HOSSIN, Mohammad; SULAIMAN, MN. A review on evaluation metrics for data classification evaluations. *International Journal of Data Mining & Knowledge Management Process*. 2015, vol. 5, no. 2, p. 1. Available from DOI: `10.5281/zenodo.3557376`.

58. RADEČIĆ, Dario. *A Non-Confusing Guide to Confusion Matrix* [online] [visited on 2021-05-02]. Available from: `https://towardsdatascience.com/a-non-confusing-guide-to-confusion-matrix-7071d2c2204f`.

59. SANTOS, Igor; PENYA, Yoseba K.; DEVESA, Jaime; BRINGAS, Pablo Garcia. N-grams-based File Signatures for Malware Detection. *ICEIS (2)*. 2009, vol. 9, pp. 317–320.

60. SCHULTZ, Matthew G.; ESKIN, Eleazar; ZADOK, F.; STOLFO, Salvatore J. Data mining methods for detection of new malicious executables. In: *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. 2000, pp. 38–49. Available from DOI: `10.1109/SECPRI.2001.924286`.

61. BEHNEL, Stefan; BRADSHAW, Robert; SELJEBOTN, Dag Sverre; EWING, Greg; STEIN, William; GELLNER, Gabriel; AL., et. *Cython - an overview* [online] [visited on 2021-05-08]. Available from: `https://cython.readthedocs.io/en/latest/src/quickstart/overview.html`.

Appendix A

Acronyms

COFF Common Object File Format

DDOS Distributed Denial of Service

DL Decision List

DLL Dynamically Linked Library

DT Decision Tree

FN False Negative

FP False Positive

FPR False Positive Rate

I-REP Incremental Reduced Error Pruning

KNN k-nearest neighbours

LR Logistic Regression

MDL Minimum Description Length

ML Machnine Learning

MS-DOS Microsoft Disk Operating System

PCA Principal Component Analysis

PE Portable Executable

RBC Rule-based classifier

REP Reduced Error Pruning

RF Random Forest

RIPPER Repeated Incremental Pruning to Produce Error Reduction

TDL Total Description Length

TF.IDF Term Frequency times Inverse Document Frequency

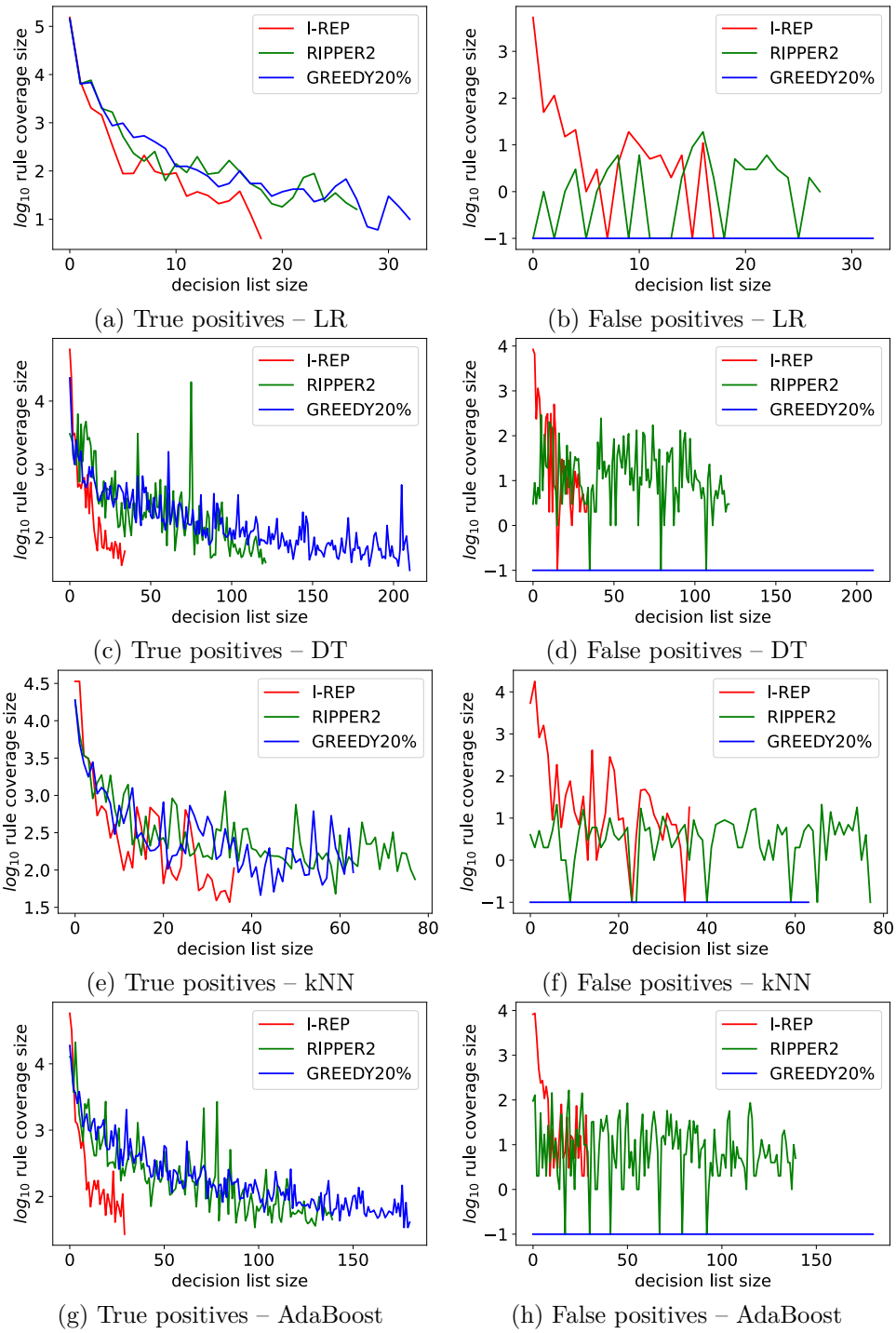
TN True Negative

TP True Positive

TPR True Positive Rate

Appendix B

Figures



(a) True positives – LR

(b) False positives – LR

(c) True positives – DT

(d) False positives – DT

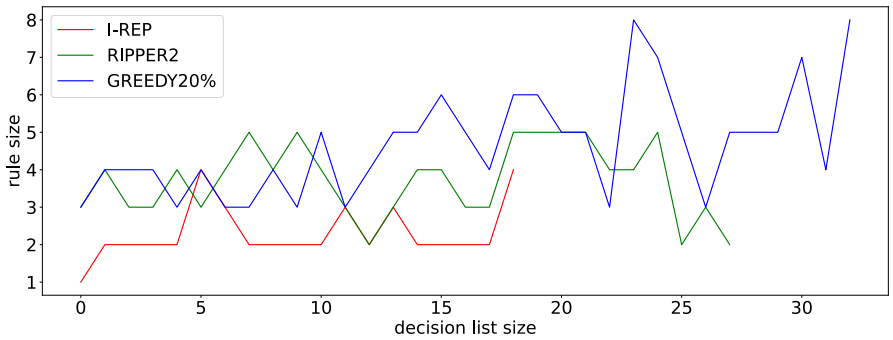
(e) True positives – kNN

(f) False positives – kNN

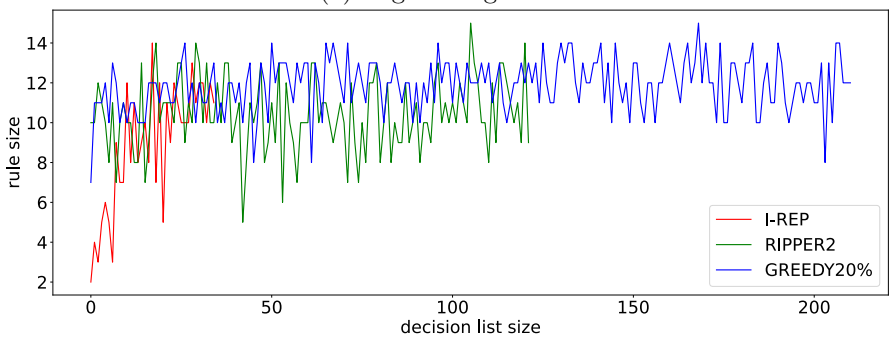
(g) True positives – AdaBoost

(h) False positives – AdaBoost

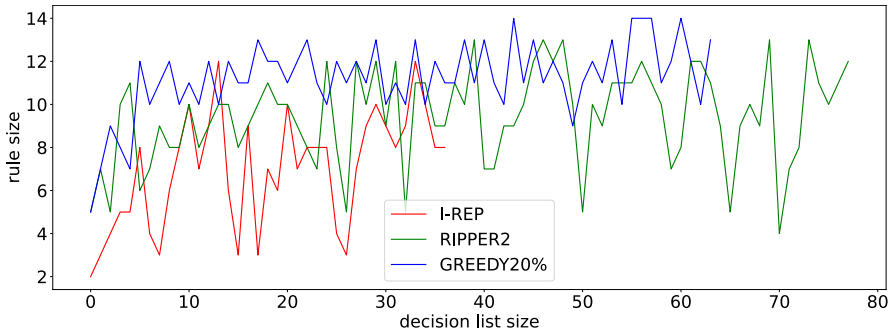
Figure B.1: Rule coverage size – EMBER



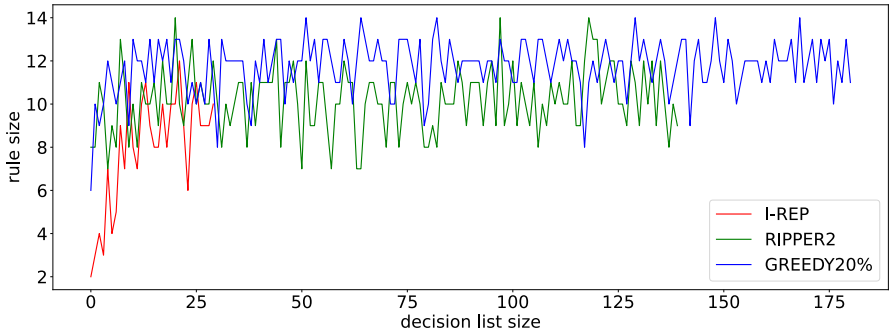
(a) LogisticRegression



(b) DecisionTree



(c) kNN



(d) AdaBoost

Figure B.2: Rule size – EMBER

Appendix C

Contents of enclosed SD Card

- `README.md` ..... Directory description
- `src`
  - `README.md` ..... src description
  - `rules_algos` ..... Implementation of RBCs & ML interpretation
  - `ml_methods` ..... ML – feature selection, transformation & training
  - `thesis` ..... Thesis text source code in LaTeX
- `text` ..... Thesis text
  - `BT_Dolejs_Jan.pdf` ..... Thesis text in PDF format
- `datasets` ..... Used datasets