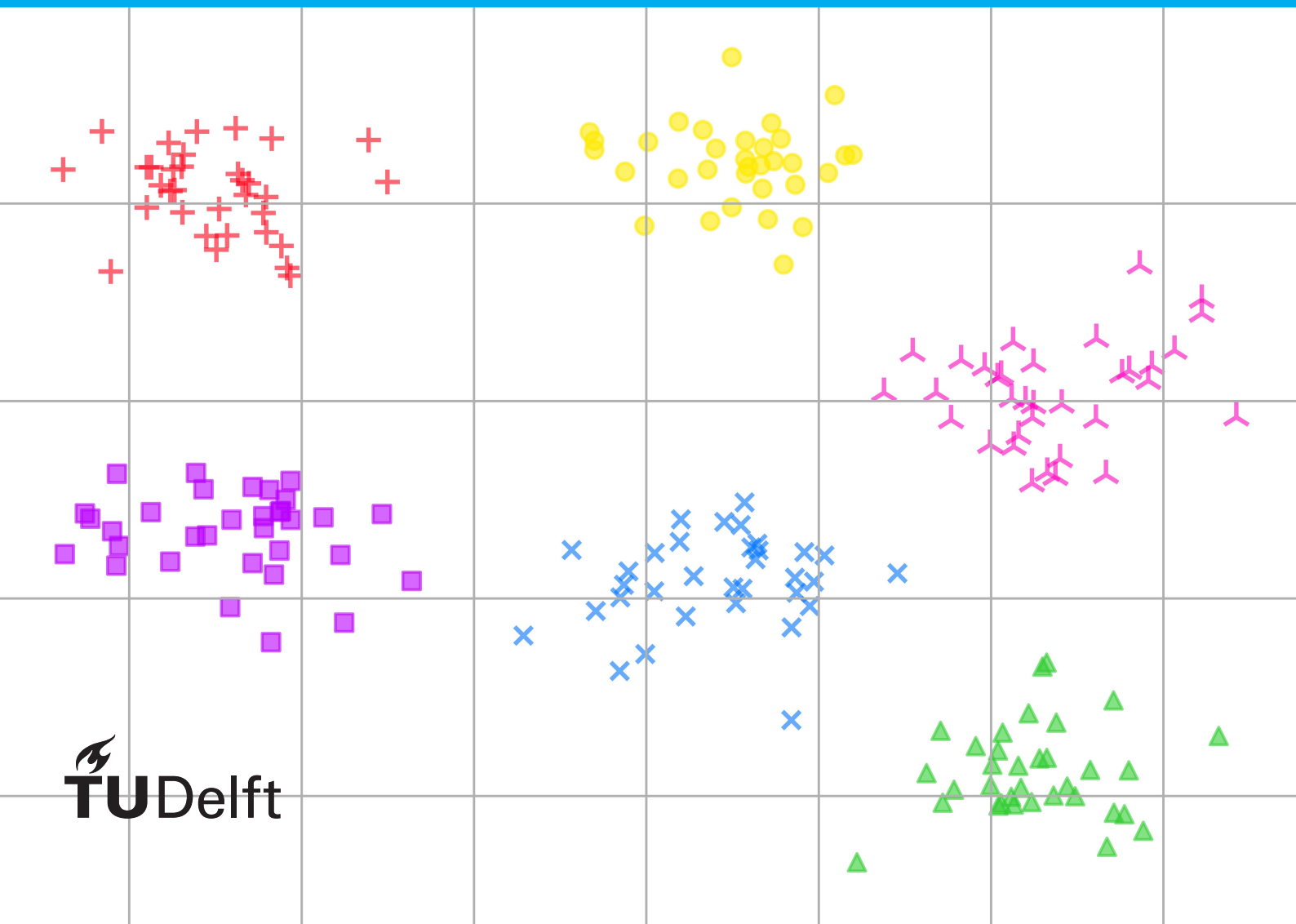
Predicting software vulnerabilities with unsupervised learning techniques

Ka-Wing Man



TU Delft

Predicting software vulnerabilities with unsupervised learning techniques

by

Ka-Wing Man

to obtain the degree of **Master of Science** in

Computer Science at the Delft University of Technology,

to be defended publicly on Thursday 20 August 2020 at 15:00.

Student number:	4330714	
Project duration:	20 May 2019 – 20 August 2020	
Thesis committee:	Dr. ir. S.E. Verwer,	TU Delft, supervisor
	Dr. A. Panichella,	TU Delft, supervisor
	Prof. Dr. R.L. Lagendijk,	TU Delft, chair

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft Delft University of Technology

Preface

There comes a time when a student has to graduate. For me that time is now, in the middle of the COVID-19 pandemic. It has been more than two years since I could start with my thesis. For most of the first year, I have struggled to find a professor who would supervise me during my research and a research topic that suits my interests and ultimately getting accepted into the company I did my thesis internship at. After I finally started with my thesis officially, little did I know that it would take very long to finally be able to start my research, that was at the end of my internship of nine months. My family and friends would bug me countless of times, asking me if I am already finished with my thesis. They can now finally witness this moment in my life. Especially my parents, they are not the youngest anymore. As education is important in the Chinese culture, they have always wanted their children to graduate with a master's degree.

Firstly, I would like to thank the people who helped me throughout this research and who I, without their help, could not have written the thesis. The first two people are my supervisors Sicco Verwer and Annibale Panichella, who have given me the opportunity to conduct a research and to write a thesis report under their supervision. They have taught me how to do research scientifically, but they have also enlightened me with their in-depth knowledge of this research topic, and have given me mental support when things did not look bright. The meetings with Sicco and all my fellow thesis students under his supervision, where I could sometimes speak out my heart about what was bugging me. The meetings with Annibale, where we would start with what pizza my partner and I ate since the last time we saw him and him telling us that we are eating biscuits, not real authentic Italian pizza with a fluffy edge. Thank you both for your effort of supervising me, your expertise and your experience in this research world.

I would also like to thank my partner Dinesh Bisseser, who I started this research together with at the same company. Our partnership goes back even before this thesis. There were numerous times that we would work non-stop on our homework through the entire night while sitting together or voice call each other. We both remember very clearly the time that we worked on some assignment on the campus until midnight, both went home and continue working on it the entire night while voice calling, finally solving the problem in the early morning and immediately met in Delft to sign off our work at 09:00. Or that one time that we worked on a project for a resit, giving up our entire week of spring break to make sure we would pass, just to fail the course in end the because a group member acted on his own during demo and accidentally broke our code, wasting our well-deserved spring break. The uncountable times that we ate Kapsalon and pizza during dinner or even after midnight. Dinesh, I am sure that you will graduate quickly after I do. I hope that our partnership will not end after these theses. However, I am certain our friendship will definitely last even after we graduate.

Not to forget Hennie Huijgens, who was a guest researcher at the TU Delft but primarily worked at ING. He arranged an internship for Dinesh and I at ING and guided us during our internship, brainstorming together with us and taught us how to act professionally in front of other ING employees that we needed to work together with for this research. Hennie, thank you that you were willing to help us whenever we needed you.

Lastly, I would like to apologize to my uncle for me taking unnecessarily long to finish my thesis and to graduate. He was fighting against his illness for the past four years and whenever I came across him in the city center of Rotterdam or whenever I visited him, knowing that he might not make it, every time he would ask me if I could already graduate, hoping that he could witness my graduation. I, however, had to answer negative every time he asked. Sadly, he passed away a few months back. My heart goes out to my aunt, his window. I wish that he could witness this moment, as he had great hopes for me that I will succeed in whatever I will do in the future.

Speaking of the future, I may return to a university to pursue a Doctor of Philosophy (Ph.D.) degree. But those ambitions rest for a few years first. I have other ambitious plans at this moment.

Ka-Wing Man
Rotterdam, August 2020

Abstract

As software is produced more and more every year, software also gets exploited more. This exploitation can lead to huge monetary losses and other damages to companies and users. The exploitation can be reduced by automatically detecting the software vulnerabilities that leads to exploitation. Unfortunately, the state-of-the-art methods for this automated process are not perfect and thus more research is needed to address this issue.
This research was partly done at ING, one of the banks of The Netherlands, in order to find a software vulnerabilities prediction method that is more efficient than their already deployed static code analysis tool Fortify Static Code Analyzer. This report proposes a method to predict software vulnerabilities in code using unsupervised learning methods. The data set is comprised of software metrics of code written by developers of ING, in conjunction with its corresponding label whether the code was vulnerable or non-vulnerable, confirmed by a security expert. Principal component analysis reduced the dimensions of the data set. From here on, the unsupervised learning technique k-means was used to build our prediction model and a distance-based anomaly detection technique was applied to find the software vulnerabilities. This produced poor results. In a final attempt to find better results, k-nearest neighbor was used to build a new prediction model and another distance-based anomaly detection technique was applied. The outcome of this latter method was surprisingly good.

Contents

1 Introduction

1.1. Introducing problem

Software has become an important part of our lives. Nowadays, even the consumers are exposed to software. Billions of people in the world are using a smartphone, a laptop or some other embedded device. These devices run on software. For example, users of smartphone take pictures for social media. Laptop users are writing. Gamers play games on their home console. These software sometimes have features that were not intended by the developer. If these features are malicious, they can be used to attack a system. Such software vulnerabilities are a big issue in the software world.

Security is one of the major factors to take into account when developing software. Software is being developed and released, now more than ever. With more applications comes more users. All of these users generate vulnerable data. This data is interesting not only for companies and organizations to process and analyze, but also for adversaries. Security and data breaches happen more often and the impact seems to be bigger every year. It is clear from the NIST study[1] that the amount of vulnerabilities in IT has been increasing the last couple of years.

Attacks can have different kind of risks. In 2018, Business Insider reports that billions of user accounts have been compromised in the top 21 biggest data breaches [23]. Adversaries can use these compromised data to sell it on the black market or blackmail people and companies. In July 2015, for instance, a group of adversaries stole the user data of dating website Ashley Madison. This website was actually used for having affairs. Adversaries could use this data to blackmail or publicly shame users, for most users would have their affairs in secret [27]. Another kind of risk, for example, is ransomware infection. The idea behind ransomware is that computers infected with a malware, having all its files encrypted in such a way that the files cannot be accessed and decrypted by the users themselves. If the users would pay the hackers within a time limit, the files would be unencrypted again and thus releasing the computer as a hostage. If the time limit is expired without payment, the files are destroyed. In May 2017, a ransomware cryptoworm named WannaCry was released worldwide and targeted computers with Windows OS [35]. Data, however, is not the only important digital asset. One of the most recent hack with big monetary losses happened in 2018. 12000 VISA cards were stolen from Cosmos Bank in India, which were used for 15000 transactions, totaling in a lost of $13.4 million [28]. One of the biggest hacks, concerning banks, is the Carbanak group. They stole almost $1 billion from over 100 banks around the world [22]. Another hack at a financial institute happened in February 2016. Hackers issued a fraudulent transfer of almost 1 billion USD from the Federal Reserve Bank of New York's account at the Central Bank of Bangladesh. Even though a total of 870 million USD was refused or halted, 81 million USD were successfully transferred[2], of which only 18 million USD has been recovered up until today[3].

These attacks are possible due to software vulnerabilities. Adversaries can exploit these vulnerabilities to gain unauthorized access to applications and data. There are many different definitions for "software vulnerabilities"

[1] https://nvd.nist.gov/vuln-metrics/visualizations/cvss-severity-distribution-over-time
[2] https://nypost.com/2016/03/22/congresswoman-wants-probe-of-brazen-81m-theft-from-new-york-fed/
[3] https://newsinfo.inquirer.net/807690/ex-rcbc-branch-manager-free-on-bail

[24]. The best definition we found for a software vulnerability according to Liu et al., and also within the scope of this research is thus:

"An instance of a mistake in the specification, development, or configuration of software such that its execution can violate the explicit or implicit security policy".

Thus, cybersecurity is becoming more important everyday.

Unfortunately, creating software also means creating vulnerabilities. More and more software is being developed and thus more vulnerabilities in software exists.[4] These vulnerabilities can form risks, some more than others, when attackers try to exploit them. These risks can harm people and companies.

To overcome these problems, vulnerabilities in software need to be tackled as soon as possible during the development of software, as bug fixes become more costly over time [38]. To ensure that most vulnerabilities are not present after release, developers have to often test and check their code for vulnerabilities. This can be done by the developers them self or by security experts. However, this process can take a lot of time and can create delays for the development of the rest of the software.

Predicting vulnerabilities can help developers check for vulnerabilities without the aid of security experts and help speed up the development process. By training a prediction model into detecting whether a certain deployment is vulnerable, developers can act accordingly.

1.2. ING, Bank of The Netherlands

This report describes the research that was conducted in collaboration with ING, the biggest bank of The Netherlands[5]. ING is a global bank with 53.000 employees over 40 countries, serving 38,4 million customer. Their product include savings, payments, investments, loans and mortgages in most of their retail markets. They provide lending, tailored corporate finance, debt and equity, market solutions, payments & cash management and trade and treasury for their wholesale banking clients, which are larger customers or organizations such as large corporate.

ING provided the penetration tests data and the corresponding source code, allowing us to use it for research purposes. At ING, there are multiple teams developing all kind of software. Whenever a team wants to deploy their software, the software has to be checked by the penetration testing team for any software vulnerability. Various tools are used for penetration testing and code reviewing the software. Depending on the risks of the vulnerabilities, certain actions must be taken before the software can be deployed.

In the previous section is explained what attackers have done in the past and what its impact was. It is needless to say that no (financial) institutes such as ING can afford such security and data breaches. Therefore, ING needs to limit the risk of having their digital assets stolen as much as possible. To ensure attackers can not exploit applications, software vulnerabilities need to be found and taken care of before applications are released. ING uses penetration testing teams to ensure that their applications does not contain software vulnerabilities.

ING makes use of two automated static code analysis tools such as CheckMarx[6] and Fortify[7] to find for potential software vulnerabilities. Usually whenever development teams have static code analysis tools deployed, they run these tools before the deployment of their software, to see whether their code needs revision because of security risks.

1.3. Problem statement

As more and more software is developed every year, the created software needs to be tested for vulnerabilities or the exploitation of such vulnerabilities could probably cost more than the testing of the software. Also, no software user wants its data to be stolen and sold by hackers on the black market. The testing of the software

[4] https://www.enisa.europa.eu/publications/info-notes/is-software-more-vulnerable-today
[5] Based on total assets:
https://www.banken.nl/nieuws/20909/ranglijst-grootste-nederlandse-banken-2018
[6] https://www.checkmarx.com/
[7] https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

is expensive because there are not enough specialized pentesters to hire and testing the software is sometimes tedious and time-consuming.

To tackle the problem of the lack of enough specialized pentesters, the testing can be partly automated which is cheaper than pentesters and faster. However, automated testing has its own problem, namely that its results need to be more reliable and they have to truly find vulnerabilities instead of returning lots of false positives.

While ING has already deployed an automated static code analysis tools, static code analysis tools' biggest flaw is the high false positives rate. Fortify is not an exception to this. The ratio of true positives with respect to all confirmed true positives and false positives, which is called precision (see Appendix A for the formulas of such classification scores), was calculated to be 0.02. It is again tedious, time-consuming and waste of resources by validating all the found potential vulnerabilities by static code analysis tools as the false alarm rate is 98%.

Machine learning is one of the methods in finding software vulnerabilities that may give good results. These results are shown in the related work section. The difference between static code analysis and a machine learning-based approach is that static code analysis interprets source code like a real compiler and knows what the code does to analyze different execution paths. A machine learning-based approach, however, does not interpret the code, but uses meta-data and statistics of the code to find software vulnerabilities.

Formally, the goal of this study is defined as:

> Developing a reliable machine learning-based process that finds software vulnerabilities, minimizing the false positives and false negatives.

1.4. Research Design

The research design is an important part of the entire study, as the design forms the backbone of the research. First, the process is outlined. Then the research questions are specified, and last, the research scope is described.

1.4.1. Research process

The setup of this research is as follows. See figure 1.1

Fortify provides us the code base, what kind of potential vulnerabilities are present and their corresponding label given by a pentester (or given by a developer but validated by a pentester) whether the potential vulnerability is truly a vulnerability or a false alarm. This data needs to be extracted. This validation of a pentester is important as most of the potential vulnerabilities are false positives. With this, each of these possible software vulnerabilities can now labeled as true positive of true negatives (as they are false positives) in our to be created data set.

This data set is used for applying machine learning techniques. This data set is constructed by software metrics. Each file extracted from the code base of Fortify is used as input for a software metric calculator tools. This tool will return a row, with the corresponding software metrics of the code, for in the data set. With each file already having labels given by the penetration testers, the data set will have columns of data in combination with a label and will be ready for cleaning and analysis.

The data set needs to be inspected, removing and altering data whenever needed. Then, a thorough analysis is done to provide us insight of the data set and giving us the opportunity to correctly interpret the data. Afterwards, we apply a dimensionality reduction technique with PCA. With a reduced data set, we cluster our data using k-means. With the clusters, we test our method by using a distance-based anomaly detection technique, in which we predict the labeling given manually by ING employees. Finally, we will discuss our results, possible future work and reflect on our work.

1.4.2. Research questions

This research is set up to recognize the pattern of the data set in order to predict the label with the data as input, which in turn helps us to detect any future vulnerable code. To predict the software vulnerabilities in

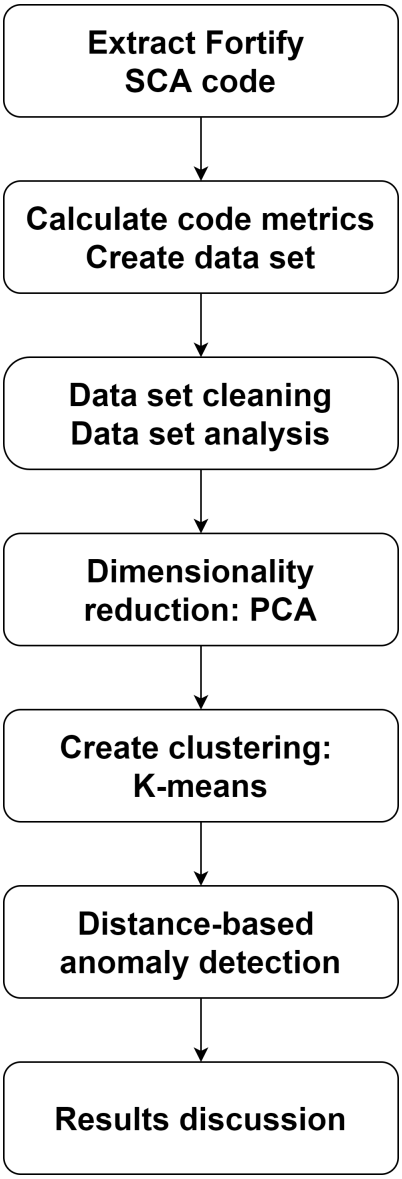


Figure 1.1: Research flowchart

an unsupervised learning setting, the *k*-means algorithm is chosen above other algorithms. This choice will be explained in the clustering chapter. The main research question that this research aims to answer, which is in line with our research goal, is:

> **RQ**: Can we predict software vulnerabilities, in an unsupervised learning setting, better than the Fortify by using the *k*-means algorithm?

To answer the main question of this research, it is broken down into three sub-questions in order to answer the main research question in a structured manner.

To have an optimal data set, a thorough data set analysis is required. First, cleaning the data set by removing unwanted data and altering data if needed. With an uncleaned and therefore non-optimal data set, one could suffer from the noise present in the data set, which is data that is actually not a part of the pattern in the data that we seek. By having the model also learn the noise, it will make predictions partly based on the noise and our model will likely overfit and therefore incorrect. By removing this noise, only data that represents the pattern that we want the model to recognized is left over, thus having the model predict the labels more correctly. Another

important part before actually building the model is analyzing the data. This is an important step to knowing how to interpret the data, that is know what the data means and how to interpret our eventual results. It is also to find any (unexpected) abnormalities in the data set that needs to be dealt, by understanding it and keeping the data as it is, altering the data or removing the data from the data set. This analysis include visualization of the data set and examination of the features. For these important steps, the first sub-question, further divided into two questions which serve the same purpose namely the analysis of the data set, is:

RQ1(a) What can we learn by analyzing the data set?

RQ1(b) What are the features of the data set and what subset of these features should be used for the prediction model?

The *k*-means algorithm produces clusters. These clusters are grouped by similarities in terms of values for each feature of each data point. This algorithm can produce different outcomes, as this algorithm requires randomization during initialization. Thus, the similarities on which the clusters are based on may be different for each executions of the algorithm. The reason why clusters are clustered in the manner that it is clustered may be wanted or unwanted, expected or unexpected, as the algorithm does not know what each feature means. It is up to the researcher to check whether the clusters are logical for our given problem, executin the clustering algorithm multiple times if necessary. Thus, it is essential to validate the produced clusters, before continuing to using them in our next step. For this problem, we derive the second sub-question as:

RQ2 How good are the clusters produced by *k*-means?

After clustering our data, we need a method to actually label the (unlabeled) test data in our test set. The chosen method is a distance-based anomaly detection method. This choice and its process will be explained in the anomaly chapter. The answer to the main research question lies heavily on the performance of this chosen method. The performance can be measured by comparing our model's predictions with the actual label that we have, ultimately allowing us to answer the final sub-question:

RQ3 To what extent are software vulnerabilities detectable using our distance-based *k*-means anomaly detection method?

1.4.3. Research scope

The scope of this project covers only unsupervised machine learning methods and only one such method is chosen, namely *k*-means. For supervised machine learning methods, we refer to the parallel study of Dinesh Bisesser. At the time of submitting this thesis report, his thesis report was not finished yet and is therefore not submitted to the TU Delft repository[8] yet. However, it will be uploaded to the repository as soon as he finished his research.

1.4.4. Report structure

The structure of the rest of the report is as follows. In chapter 2, the related work of similar researches conducted by other researchers in the past is described, together with background information needed in this research and the research gap is identified. In chapter 3, the creation of the data set for this research and its analysis is explained. In chapter 4, the clustering of the data using the unsupervised learning algorithm *k*-means is shown. In chapter 5, the distance-based anomaly detection technique is outlined and its results is discussed. In chapter 6, the answers to the research questions are given, including a discussion/reflection of this research.

[8] https://repository.tudelft.nl/

1.5. Summary

As software are being exploited more and more, making huge damages to users and companies. This can be mitigated by removing software vulnerabilities. Trained pentesters are the best suited people to do this job, but their number on the market are scarce. Finding software vulnerabilities also takes a lot of time. This problem can be remedied by automating this process. ING has deployed Fortify, a static code analysis tool, but it is very limited. In order to help ING with their software vulnerabilities problem, we are setting up a research to detect software vulnerabilities using unsupervised learning techniques. The goal of this study is to goal of this study is to develop a reliable machine learning-based process that finds software vulnerabilities, minimizing the false positives and false negatives.

2

Literature review

Software vulnerability detection, also known as **software fault prediction** is a significant research area within the software engineering field for over 30 years [37]. These researches apply machine learning techniques, static code analysis and others. Among the researches that utilizes machine learning techniques, different studies uses different techniques and different authors argues some techniques are superior to others. For example, there are studies that use different machine learning models (Random Forest, Naive Bayes, Logistic Regression, Decision Tree etc.) and there are studies that use different metrics (Halstead, McCabe, Object-oriented etc.). Also, whether a model is decently trained depends on what performance metrics is chosen e.g. Area Under the Curve (AUC), precision, recall etc. Therefore, the objective of these researches is to create models with the aforementioned techniques, using one of multiple of these software metrics and also one of multiple of these performance metrics to find latent relations and patterns in order to detect software vulnerabilities.

In this chapter, the related work of software vulnerability detection research previously done by other researchers is outlined. Afterwards, some background information of techniques related to this research is explained.

2.1. Related Work

Software vulnerability detection comes with different approaches. One approach is to approximate the number of vulnerabilities. For example, Compton et al. make use of statistical analysis of McCabe and Halstead metrics to calculate the defect density [13]. This research will be based on the classification approach, that is to classify given certain software modules as vulnerable or non-vulnerable.

As there are many research paper about the topic of software vulnerability detections, there are few literature review papers about it. Radjenović et al. found 106 papers published between 1991 and 2011 [43]. They found a software metrics distribution of object-oriented metrics (49%), Halstead and McCabe's software metrics also known as traditional software metrics (27%) and process metrics (24%) while Chidamber and Kemerer's object-oriented metrics were used the most frequently. They report that the OO-metrics and process metrics were the most successful metric in finding vulnerabilities.

Catal (et al.) has two review papers on the detection of software vulnerability. In the first paper, published in 2009, they reviewed 74 papers since 2005 up until 2009. [10]. In the second paper, published in 2011, Catal found 90 papers between 1990 and 2009 [9]. He showed that more and more research is done over the years on detecting software vulnerabilities. His numbers can be seen in table 2.1. He found that the usage percentage of using public data sets and the usage percentage of machine learning algorithms both have increased slightly since 2005, and method-level metrics are still the most dominant metrics, while machine learning algorithms are still the most popular methods.

Malhotra identified 64 primary studies between 1995 and 2013 [26]. They found papers using various machine learning algorithms, like decision trees, Bayesian learners, support vector machines, neural networks etc. 49% of the 64 papers used some kind of feature reduction technique, with co-relation based feature selection as the most popular technique. 34 papers used traditional software metrics while 18 others used Chidamber and

Years	Amount of papers found
1990-2000	10
2000-2003	14
2003-2005	15
2005-2007	17
2007-2009	34

Table 2.1: Distribution of papers over the years

Kemerer's software metrics. They report that Chidamber and Kemerer's metrics Coupling Between Objects (CBO), Response For a Class (RFC) and Lines Of Code (LOC) are highly useful, but Number of Children (NOC) and Depth of Inheritance Tree (DIT) are not. These metrics are further explained in the Background Information section. They also report that more than 40 papers used recall as their performance metrics, while 30 used accuracy and more than 20 used precision also around 20 used AUC.

Haghighi et al. compared 37 different classification algorithms over 5 public NASA data and figured that Bagging has a best overall performance (measured in ACC and AUC) in fault detection systems than the other tested classifiers [16].

2.1.1. Prediction models

Different models have been proposed. Each of them have their own advantages and disadvantages. Catal et al. argued that these machine learning algorithms should continued to be used as they build better fault predictor [10].

Chowdhury et al. investigated extensively what the relationship is between complexity, coupling and cohesion (CCC) metrics and software vulnerabilities have [12]. The goal was to investigate whether complex, coupled and non-cohesive software modules have more software vulnerabilities and which CCC-metrics can be used to indicate these vulnerabilities in software. Due to different metrics being available at different development stages, the correlation was further explored to determine whether code-level (available after coding) or design-level (available after design phase) of CCC-metrics are better indicators of vulnerabilities. A case study was conducted on software vulnerability data of 52 Mozilla Firefox releases that was developed over a course of four years. A set of Chidamber-Kemerer metrics that measure complexity, coupling and cohesion was selected to analyze the correlations with vulnerabilities on design-level while mostly McCabe metrics were selected to measure the correlation on code-level. Each vulnerable software module, may it be functions, files, classes etc., were traced using vulnerability reports, bug repositories and software version archive, and for each of these modules, the CCC-metrics were calculated. Their results show that complexity, coupling and cohesion all have a correlation with vulnerabilities at a statistically significant level. Code-level metrics are deemed more strongly correlated to vulnerabilities due to their belief that code more closely represents the operational behavior of the software than the design specification, on assumption that programmers do not always follow the design specification and therefore code sometimes diverges from what is specified in the design.

Tim Menzies et al. concludes that it is irrelevant to debate which static code attributes are most successful as predictors of software defects [32]. They argue that minor changes in data (such as a slightly different sample used to learn a predictor) can make different attributes appear most useful for defect prediction. Hence, the so-called "best attributes" used for defect prediction vary from data set to data set and rather than labeling a particular subset of possible attributes as the best attributes, one should find a subset of all available attributes that is most appropriate for a particular domain (e.g. projects). Moreover, they show that the choice of learning method is far more important than which subset of the available data is used for learning. They have experimented with prediction models such as Naive Bayes (with log filters), J48, a Decision Tree generator developed by Ross Quinlan [42], and OneR, which builds prediction-rules using one or more values from a single attribute. Results show that NaiveBayes with log filters have the best average results, namely $pd = 0.71$ and $pf = 0.25$.

However, as satisfied as Tim Menzies et al. are with the abovementioned results, Zhang and Zhang argued that in [32] the models are impractical [49]. While a high pd and a low pf looks fine at first hand, they do not necessarily lead to accurate models with high precisions due to the data set being highly imbalanced. An accuracy

measure widely used in Information Retrieval is precision, which was very low in the models. For one data set tested by Menzies et al., the precision was $0,2064$, meaning if a module is predicted as defective, the probability of it actually being defective is only $20,64\%$. The precision of another data set was only $0,0202$. According to the authors, applying such models would defeat the very purpose of defect prediction, which is about allocation of limited QA resources more efficiently, so that efforts can be concentrated on the potentially problematic modules and thus the models would not be satisfactory for practical use and should therefore be improved. Tim Menzies et al responded that precision instability is the reason that they do not assess performance in terms of precision. [31].

Tim Menzies et al. proposes a method called SEVERIS, which assists a test engineer in assigning severity levels of risks of defects found [30]. This method is useful for non-experienced test engineers who might assign a lower severity level than in reality. After an issue has been raised, the test engineer takes notes of the issue and assigns it some severity level. SEVERIS trains a prediction model from past notes and a human-assigned severity level. The prediction model generates a score of how much self-confidence a supervisor has in the SEVERIS' conclusions and determines a second severity level for each issue risen. If SEVERIS' severity level differs from the test engineer's, a human supervisor can decide to review the severity level assignment of the test engineer. To support this process, the supervisor can review the self confidence score to decide if they trust the SEVERIS' assignment.

Alves et al. built a data set containing software metrics from functions, classes and files of five projects that relevant from a security point of view, as they are widely used and exposed to attacks [5]. These five projects are *Mozilla*, *Linux Kernel*, *Xen Hypervisor*, *httpd* and *glibc*. In total, there were 2875 security patches found over the five projects and 5750 snapshots of the projects were made from the corresponding repository, one commit immediately before the patch and one commit immediately after the patch. With the obtained code, the commercial software Understand[1] was used to compute the software metrics for all elements in the snapshots. The most relevant conclusions of this research was that software metrics computed by Understand and software vulnerabilities are indeed correlated. However, none of the individual computed software metrics indicate directly which function will have more vulnerabilities and therefore using predictive models in machine learning that combine multiple metrics is suggested.

Shin et al. evaluated whether code complexity and developer activity metrics are discriminative metrics that can be used to prevent software vulnerabilities and evaluated whether code complexity, code churn, developer activity metrics can predict the location of vulnerable code. Two empirical studies were set up on Mozilla Firefox and the Linux kernel as distributed of the Red Hat Enterprise Linux. In a time span of four years, 197 vulnerability reports were collected from 34 releases of Mozilla Firefox, in which a total number of 1197 instances of file changes to fix vulnerabilities observed. For Red Hat Enterprise Linux, 192 vulnerability reports are collected of one release, in which a total number of 258 instances of file changes to fix vulnerabilities were found. The code complexity, code churn, developer activity metrics were computed and analyzed.

Czibula et al. proposes a classification model that is based on mining relational association rules, which are particular type of association rules that describe numerical orderings between attributes that occur frequently in a data set [15].

2.1.2. Prediction software metrics

Back in 1970's, a critical problem software engineering had was how to modularize software such that it would be easily testable and maintainable [29]. One of the practice used back then was to limit programs by their physical size in order to ensure modularization. This method does not suffice, as 25 consecutive if-then statements could result in $2^{25} (\approx 33,5$ million) distinct control paths. One could imagine that not all of these paths would be ever tested. Thomas J. McCabe made an effort to develop a mathematical technique that will provide a quantitative basis for modularization and allows developers to identify software modules that are hard to test or to maintain. They are explained in the next section, together with Halstead's complexity measure and Chidamber and Kemerer's object oriented metrics.

There are also process metrics, these are code delta (difference between two builds in number of lines), code churn (sum of lines added, deleted and modified), the number of developers, the number of past faults, the

[1] https://scitools.com

number of changes, the age of a module and the change set size [43] [4]. Shin et al. evaluated these metrics and show that their results indicate that they are discriminative and predictive of software vulnerabilities [45].

2.2. Background information

This section described the information needed in our research. Most of these techniques are applied directly

2.2.1. McCabe's cyclomatic complexity

Let the flow of a program be represented by a control graph G with n vertices, e edges and p connected components. The graph is constructed depending on the flow of the program. The **cyclomatic complexity** V(G) of G is defined as $V(G) = e - n + 2p$

In general, it holds that a collection C of control graphs with k connected components equals to the summation of their cyclomatic complexity. This method can be used to calculate the complexity of a collection of programs, where for example subroutines exists. To prove this, let C_i with $1 \leq i \leq k$ denote the k distinct connected components, with n_i and e_i respectively be the number of nodes and edges of the i-th connected component. Then

$$v(C) = e - n + 2p = \sum_{1}^{k} e_i - \sum_{1}^{k} n_i + 2k = \sum_{1}^{k} (e_i - n_i + 2k) = \sum_{1}^{k} v(C_i)$$

A graph is **strongly connected** if all nodes can be reached from any other node.

> **Theorem 1** *If a graph is strongly connected then the cyclomatic complexity V(G) equals the maximum number of linearly independent paths v(G).*

McCabe's experiment had programmers instructed to calculate the complexity as they created software modules. McCabe noted what when the complexity was greater than 10, the programmers either split their code into subfunctions or rewrote the code. Some programmers had very distinct coding style, e.g. several loops in sequence, in which the control flow graph of their programs had similar patterns and also had high complexity. Furthermore, a close correlation was found by the project members between the ranking of subroutines by complexity and a ranking by reliability. The conclusion is that a higher complexity results in less reliable code. Therefore, the strategy to keep programs testable and maintainable is to set an upper limit to the number of linearly independent paths v(G). McCabe suggested the upper limit to be 10.

2.2.2. Halstead's complexity measure

Halstead introduced its metrics back in 1977 [17]. Measurable properties of any expression of any algorithm include the following:

η_1 = number of distinct operators
η_2 = number of distinct operands
N_1 = total number of operators
N_2 = total number of operands
Program vocabulary: $\eta = \eta_1 + \eta_2$
Program length: $N = N_1 + N_2$
Volume: $V = N \times log_2 \eta$
Difficulty = $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
Effort = $E = D \times V$

Halstead came up with an estimate for the number of bugs in an implementation:

Number of delivered bugs = $B = \frac{E^{\frac{2}{3}}}{3000}$

2.2.3. Chidamber and Kemerer's Object Oriented metrics

Chidamber and Kememrer came up with next software metrics suitable for object oriented programming languages such as Java in 1994 [11].

Its first proposed metric is the Weighted Methods per Class (WMC). Consider a Class C that has methods $M_1,\ldots,M_n$, let $c_1,\ldots,c_n$ be the static complexity (it is not defined in the original paper what the static complexity is) of those methods, then the WMC is defined as

$$WMC = \sum_{n}^{i=1} c_i$$

The WMC is a complexity measurement and it indicates how much time and effort it takes to develop and maintain an object.

Its second proposed metric is the Depth of Inheritance Tree (DIT). This is the number of ancestor classes and it measures how many of the ancestor classes can potentially affect the class in question as deeper trees have greater design complexity since more methods and classes are involved.

The third proposed metric is the Number of Children (NOC). This is the number of immediate sub-classes of the class in question. As the class can affect all its children, a class with large number of children requires more testing.

The fourth proposed is the Coupling Between Objects (CBO). It counts the number of non-inheritance relations with other classes. In their viewpoint, excessive coupling between objects outside of inheritance makes the design less modular and hard to reuse. This measure could be useful in determining how complex testing certain classes are. The more a class is coupled, the harder it is.

A fifth proposed metric is Response for a Class (RFC) is the set of methods called by any method of a class. The reasoning behind this is if a larger number of methods are invoked, the higher the complexity is of the object of that class.

The sixth and final proposed metric is Lack of Cohesion in Methods (LCOM). Let C be a class with $M_1,\ldots,M_n$ methods. Let I_i be the set of instance variables used my method M_i, then the LCOM is defined as the number of disjoint sets formed by the intersection of $I_1,\ldots,I_n$. Cohesiveness in methods is desired, as it promotes encapsulation of objects. If there is a lack of cohesion, then the class in question should probably be split into two or more classes.

2.2.4. Curse of dimensionality

Aggarwal et al. explored the behavior of the L_K-norm [3]. Their research suggests that the L_K-norm might be more suitable for $k = 1,2$, rather than $k \geq 3$. It also provide considerable evidence that the higher the dimensions are, the less meaningful the L_K norm is for higher values of k. They concluded that given a problem with a high dimensionality, a lower k may be preferred, which would be $k = 1$ (L_1-norm aka Manhattan Distance), followed by $k = 2$ (L_2-norm aka Euclidean Distance). With this idea of preferring a lower k, they presented the fractional distance metric, for which $k < 1$. In their results, they have shown that this metric is effective in preserving meaning when comparing distances. This fractional distance metrics was tested with synthetic and real data. The chosen dimensionality was 20. Using the k-means algorithm, using the fractional distance metric had a classification rate of approximately 99% with f=0.3, in contrast to 89% using Euclidean distance.

2.2.5. k-means

k-means is a well-known clustering algorithm that is used since it was independently published by different researchers. Its popularity is thanks to the algorithm being simple, effective and easy-to-implement [20].

The steps of the algorithm is as follows:

1. Choose of number of k clusters to partition n data points in.
2. Initialize k cluster centroids randomly by selecting points in the space as the data points.

3. For each data point, compute the error to each cluster centroid. Assign the data point to the cluster of the cluster centroid with the minimum error.
4. For each cluster, compute the mean of all the data points in the cluster. This mean is the new cluster centroid.
5. Repeat steps 3 and 4 until the new cluster centroids do not change.

Formally speaking, the k-means algorithms minimizes the sum of squared errors

$$J = \sum_{k=1}^{K} \sum_{i=1}^{n} (x_i - c_k)^2$$

where $c_1, ..., c_k$ are the cluster centroids and $x_1, ..., x_n$ are the data points.

2.2.6. Principal Component Analysis

One of the popular dimensionality reduction techniques is Principal Coponent Analysis (PCA). PCA is a multivariate analysis method first introduced by Karl Pearson in 1901 ([41]) and then independently developed by Harold Hotelling in 1933 ([19]) [21]. The idea behind PCA is the reduce the dimensionality of a data set, that has a high number of correlated features, while preserving as much as possible of the variation of the data set. This reduction happens by transforming the original data set into a new set of features which are called principal components. These principal components are linear independent (hence uncorrelated) and are ordered descending so that the first principal component retains most of the variation present in all of the original features. This results in the first principal component being the most significant while the last being the least significant.

The principal components transformation can be performed using singular value decomposition (SVD) [48] or by using the covariance. The covariance method works as follows [46]: Let an n rows $\times p$ columns matrix $\mathbf{X}$ denote the data set. Let row vectors $\mathbf{N}_1, ..., \mathbf{N}_n$ denote each row of $\mathbf{X}$ and let column vectors $\mathbf{P}_1, ..., \mathbf{P}_p$ denote each columns of $\mathbf{X}$

1. Construct $\bar{\mathbf{X}}$ by calculating the mean $\bar{p}_i$ of each column $\mathbf{P}_i$ with $i = 1, ..., p$, then subtract each element of $\mathbf{P}_i$ with the mean $\bar{p}_i$
2. Calculate the covariance matrix $\mathbf{M}$ of $\bar{\mathbf{X}}$.

   The covariance matrix $\mathbf{M}$ is a $p \times p$ matrix, where the i-th rows and j-th column denote the covariance of $\mathbf{P_i}$ and $\mathbf{P_j}$ with $i, j = 1, ..., p$.

   Let $\mathbf{A}$ and $\mathbf{B}$ be two column vectors of $\bar{\mathbf{X}}$. Let $\mathbf{A}_i$ and $\mathbf{B}_i$ be the i-th element and $\bar{a}$ and $\bar{b}$ be the mean of $\mathbf{A}$ and $\mathbf{B}$ respectively. Then, the covariance of $\mathbf{A}$ and $\mathbf{B}$ is calculated with the following formula:

$$cov(\mathbf{A}, \mathbf{B}) = \frac{1}{n-1} \sum_{i=1}^{n} (\mathbf{A}_i - \bar{a})(\mathbf{B}_i - \bar{b})$$

3. Calculate the eigenvectors and their corresponding eigenvalues of the covariance matrix $\mathbf{M}$, i.e. find all vectors $\mathbf{v}$ and scalar values λ such that $\mathbf{Mv} = \lambda\mathbf{v}$.
4. Order the eigenvalues from highest to lowest. Construct a feature vector $\mathbf{E}$ by concatenating of the first d eigenvectors, with d the dimensions you choose to reduce your original dataset to, i.e. $\mathbf{E}$ = (eigenvector$_1$, ..., eigenvector$_d$). Note that $\mathbf{E}$ is a $p \times d$ matrix
5. To get the principal components matrix $\mathbf{X}_{PCA}$, multiply the transpose of $\mathbf{E}$ with the transpose $\bar{\mathbf{X}}$, i.e.

$$\mathbf{X}_{PCA} = (\mathbf{E}^T \times \bar{\mathbf{X}}^T)^T$$

By making this transformation, the original data is represented on the eigenvectors instead of their original axes. The intention of representing it on the eigenvectors is because eigenvectors are orthogonal (perpendicular) if they exist and the transformation is the most efficient if the eigenvectors are orthogonal. By leaving out the last principal components, some of the information is lost while most of the information is still kept.

2.2.7. t-Distributed Stochastic Neighbor Embedding

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a dimensionality reduction technique that is used for visualizing data sets with high dimensions [25]. It minimizes the divergence between two distributions: a distribution that measures pairwise similarities of the input objects and a distribution that measures pairwise similarities of the corresponding low-dimensional points in the embedding[47]. t-SNE works well at creating a map that reveals structures at many different scales, the most important one for this research being that data points that are close to each other in a high dimensional level, can be shown close to each other in a low dimensional level (2D or 3D). An excellent visualization example with t-SNE can be seen in figure 2.1, while other tested visualization methods failed to show the clusters grouped together. Its advantage over PCA is that PCA is a linear projection, while t-SNE can deal with both linear and non-linear data. t-SNE is a rather complex algorithm and we won't go into the details of this algorithm. What is important to note is that t-SNE's main parameter is the *perplexity*. The perplexity can be seen as a smooth measure of the effective number of neighbors. The performance of t-SNE is fairly robust to changes in this parameter. Suggested values of perplexity are between 5 and 50. The original papers states that t-SNE is suitable for visualization by dimensionality reduction, but it is not clear how the dimensionality reduction performs if the goal is to only reduce dimensions. Hence, the purpose of t-SNE in this research is only visualization.

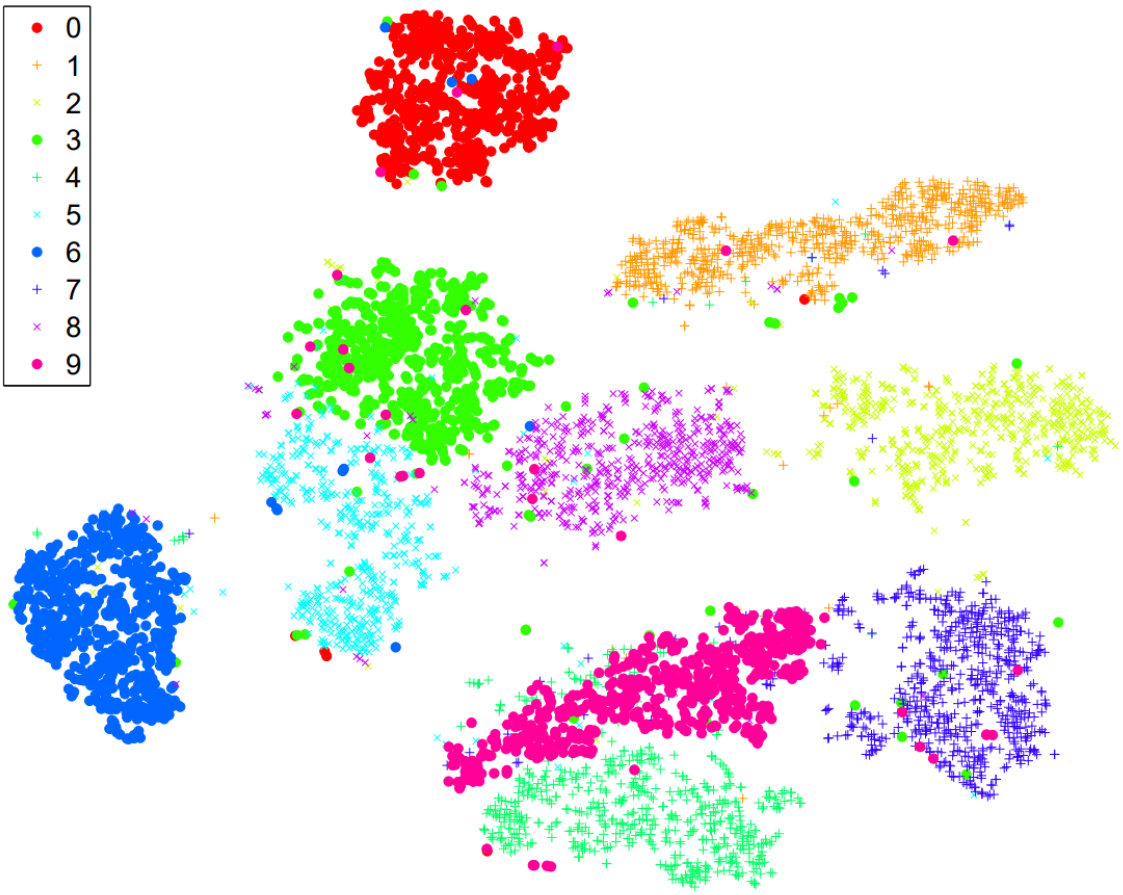


Figure 2.1: A visualization of 6000 handwritten digits from the MNIST data set from the original paper. Here you can see that clearly that 10 clusters are grouped together.

2.3. Research Gap

As shown in the related work section, lots of research have already been done in the past, different methods have good results, but the same methods do not necessarily work for different data set than that are described in the research papers, thus the different methods that had good results in the prediction of software vulnerabilities might

not work on the data available at ING. Also, to the best of our knowledge, the software vulnerability data to be extracted have not been ever (extensively) examined before, how good it works with machine learning techniques to predict software vulnerabilities. To fill this research gap, we apply known machine learning techniques, which have worked in the past, to the software vulnerability data available at ING.

3

Data set

3.1. Micro Focus® Fortify Static Code Analyzer

Micro Focus® Fortify Static Code Analyzer, in this report mostly shortened as Fortify, is one of the commercial static code analysis tools deployed at ING. It is a closed-source software that was acquired by Hewlett-Packard (HP) and later spun off and merged with a British software and consulting company Micro Focus [44]. Fortify can be integrated in CI/CD tools, for instance IDE plugins for Eclipse, Virtual Studio etc. and bug tracker tools such as Jenkins, Jira, Atlassian etc., which is useful for real-time analysis and results when developing. It detects 788 unique categories of vulnerabilities across 25 programming languages (including popular languages like Java, Python, C/C++ and C#) and spans over 1.007.000 individual APIs. Micro Focus claims that the accuracy has a true positive rate of 100% in the OWASP 1.2b Benchmark as demonstrated. (Author's note: yet the false alarm rate we found for all sorts of vulnerabilities is 98% as stated in chapter 1. It shows that their claim is not that spectacular.)

3.1.1. Inner workings

Fortify uses multiple algorithms and a knowledge base of secure coding rules to analyze source code for vulnerabilities that might be exploited. It analyzes every execution path to identify and rectify vulnerabilities. Its analysis is done by reading the source code, much like a compiler, and converts them to an intermediate structure, which is used to locate the vulnerabilities. This fits exactly in the context of a general static code analysis [6, 8]. The exact algorithm and knowledge base is unknown as Fortify is a commercial product [34]. The knowledge base can be expanded by including custom rules. Fortify consists of eight vulnerability analyzers. Each of these analyzers evaluate the source code to check whether rules specific to the analyzers are violated. The rules are to recognize aspects in the source that could follow in a certain security vulnerabilities [33].

The **Buffer Analyzer** checks the source code and determine using limited interprocedural analysis whether reading or writing beyond the allocated space of a buffer in the memory, in which the program is loaded, is possible. If so, the code is vulnerable to buffer overflow attacks, where parts of the memory can be overwritten to change the flow of the program or to even inject malicious code [14]. The **Configuration Analyzer** looks for weak policies in configuration files. The **Content Analyzer** does in static HTML files and also files that contains dynamic HTML i.e generates HTML such as PHP, JSP and classic ASP files. Security misconfiguration is in the top 10 of most critical web application security risks (OWASP Top 10 2017) [40]. The **Control Flow Analyzer** searches for possible dangerous sequences of operations and determines whether a set of operations are executed in the same order as intended. Examples are initializing variables before using them or utilities such as XML readers are configured correctly before being used. The **Dataflow Analyzer** searches for potential vulnerabilities that utilizes tainted data for potentially dangerous use, such as an user-controlled input string of unbounded length that is copied into a statically sized buffer or a user-controlled input string that is used to inject SQL code. The **Null Pointer Analyzer** speaks for itself. In certain programming languages such as C, C++ and Java, the danger of a null pointer is that when a NullPointerException is raised, an attack may be using that exception to bypass security logic or causing the application in revealing debugging information that might be

useful in planning the next attack [39]. The **Semantic Analyzer** finds potentially dangerous uses of methods and API, such as deprecated methods in Java and the `gets()` method in C/C++ that can be ussed for buffer overflows. Last, the **Structural Analyzers** looks for potentially dangerous flaws in the structure of the code. It tries to identify violations of secure programming practices and techniques that are too complex to inspect. An example is searching flags of instances of dead code that is never executed due to some predicated being always false.

3.1.2. Navigating through the app

After logging in on Fortify, the dashboard is the first thing you see. It has a list of projects that have completed scans in the past. Each project is displayed with the statistics as shown in figure 3.1

Statistic	Meaning
Files Scanned	Total number of files scanned to date for this application version.
Lines of Code	Total number of lines of code scanned to date for this application version.
Bug Density	Number of possible issues found per 10.000 lines of source code scanned.
Open Issues	Total number of issues found for this application version that have not been remediated.
Issues Pending Review	Total number of issues for this application version that have not been reviewed.
Avg Days to Review	Average number of days it took to review issues found for this application version.
Avg Days to Remediate	Average number of days it took to remediate issues found for this application version.

Table 3.1: The dashboard statistics on Fortify explained.

Fortify works as follows. See figure 3.2 for a typical flowchart of the usage of a deployed static code analysis tool. The projects are scanned for vulnerabilities. A project may takes hours depending on the size of the project. After the scan is complete, a table for each project is set up. Each row represents a project. The columns consists issue name, the location (i.e. file name and line number), analysis type, criticality and the tag.

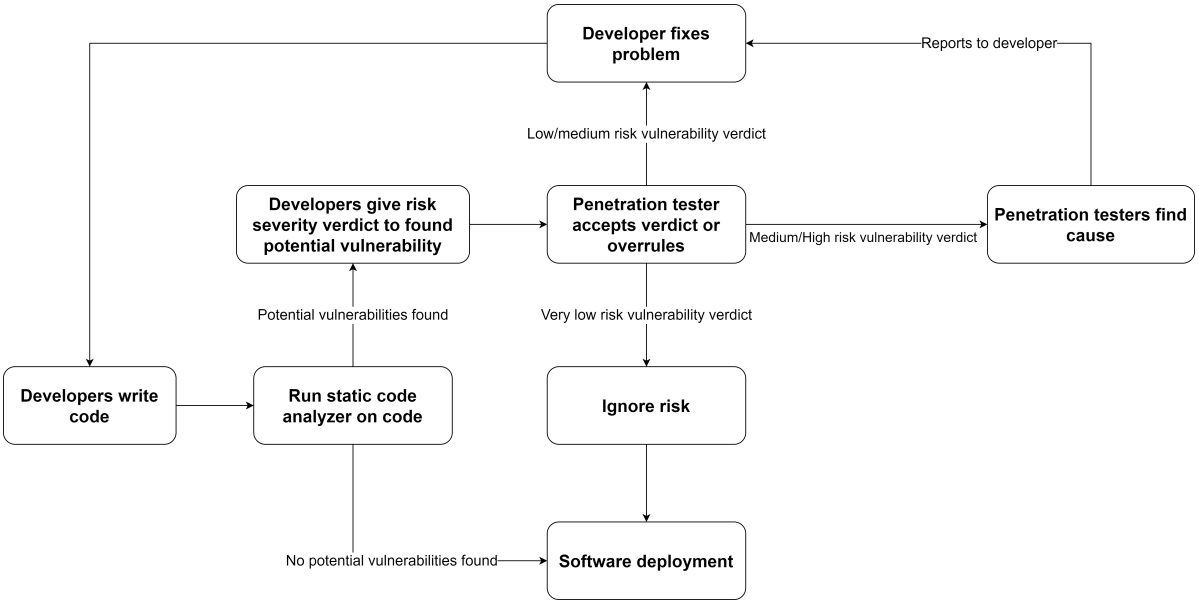


Figure 3.2: Typical flowchart of the usage of a static code analysis tool.

Issue name is the type of vulnerability, e.g. SQL injection or usage standard pseudo-random number generators that are not secure against cryptographic attacks. The primary location consists of a file name and line number of the vulnerability. Analysis type is mostly SCA, which stands for static code analyzer. A very few of the

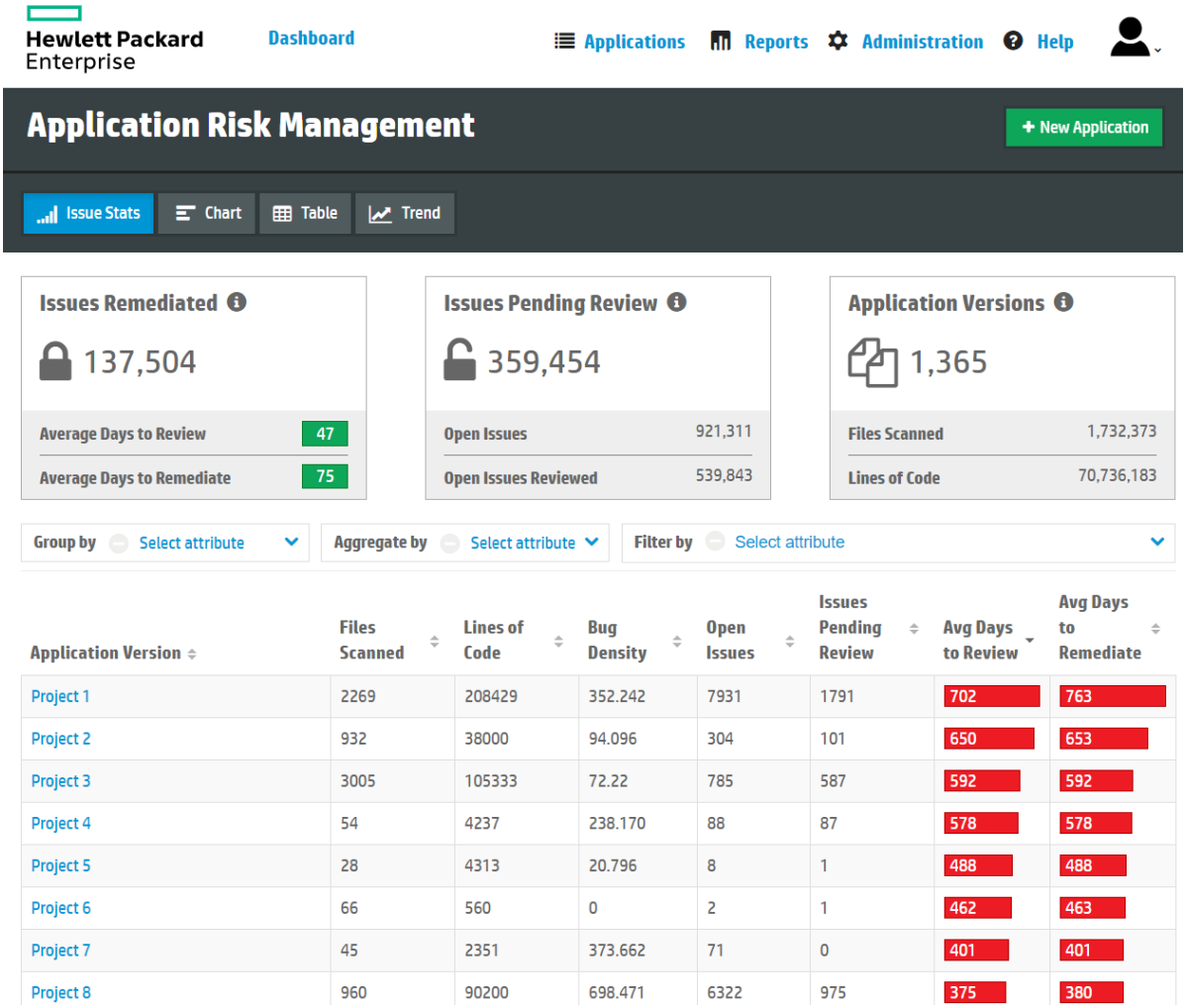


Figure 3.1: The dashboard of Fortify with dummy names and numbers. Note: The *Todo List* and *Acitivity Feed* on this dashboard page are removed from this image for readability and scalability purposes.

vulnerabilities are CUSTOM. This is only when vulnerabilities are found manually by an employee and manually added to the list of vulnerabilities i.e. it is a True Negative vulnerability. CUSTOM-labeled vulnerabilities are shown in the list of vulnerabilities, but when clicked on, the code[1] and comments section won't show as it displays the message "Unexpected error has occurred. Please contact your administrator". Ultimately, it was decided that this data could not be used.

Criticality is an auto-generated label that suggests whether the level of severity of the vulnerability is *low*, *medium*, *high* or *critical*, with critical being that the vulnerability needs to be fixed immediately and low being that the fix can wait. The tag are five possible labels denoted by *not an issue*, *bad practice*, *reliability issue*, *suspicious*, *exploitable*. The tag is set by the users. After the security engineers review the decisions of the developers, they can either set the vulnerability to *approved* or *not approved*, depending on whether they agree with the developer. Any potential vulnerability that has been reviewed, should be suppressed (see bottom-right in figure 3.4). Vulnerabilities that are suppressed are not visible on the web app in the list of vulnerabilities anymore, but they are still in the database and can be retrieved.

When clicked on a row as can be seen in figure 3.3, the code, an explanation of the given vulnerability type and a general fix of the given vulnerability type is shown. This gives you a possibility to look into the file of the vulnerable code without having access to the entire project.

[1]Every mentioning of the code in this chapter means only the file where the supposed vulnerability lies, not the code of the entire project.

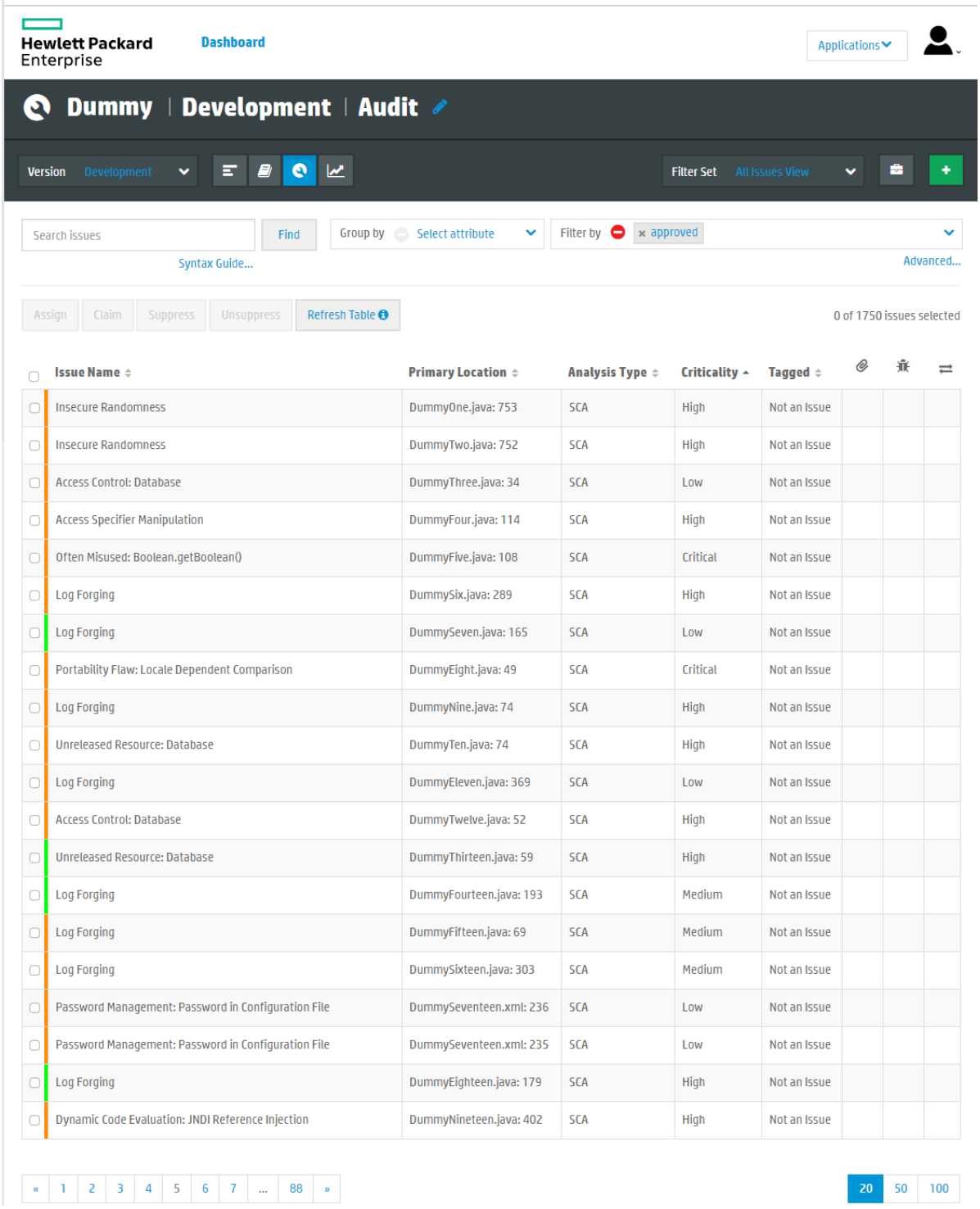


Figure 3.3: List of vulnerabilities for some project with dummy names and numbers.

Each and every vulnerability needs validation whether it is a true positive or a false positive. Due to the scarcity of security engineers in general, the vulnerabilities can be first checked by the developers of their respective projects and then checked by the security engineers so latter's time. These developers can assign a vulnerability with *approved* if it is deemed a false positive, or *not approved* for a true positive. They can also leave a comment, explaining their choice. Security engineers double-check the assignment of the vulnerability. As they have more security knowledge than the average developer, they sometimes overrule the assignment. For example, if they think the vulnerability is not a false positive, they change the assignment from approved to not approved, and

they will also leave a comment why the assignment is changed. A history of assignments and comments with their timestamps can be displayed for a single vulnerability or a set of vulnerabilities, if multiple vulnerabilities have the same issue name and are in the same file.

3.1.3. Data extraction considerations

Unfortunately, any useful insight of the vulnerabilities per project is not available and extracting the data from these tools on their website is not easy. For example, the dashboard of Fortify does not show any information to determine the number of the vulnerabilties in the project. Also, Fortify does not seem to have a useful export feature. Nevertheless, going through all the projects manually is tedious and it might takes weeks to extract the data.

After speaking with the administrators of Fortify at ING, the data could be exported in two ways. By performing an SQL export of the data with help of an administrator of Fortify or by using the RestAPI of Fortify. These methods were only possible in theory, but in practice both were not possible.

Extracting vulnerabilities, comments, code etc. of 930 project with an SQL export uses a significant amount of resources, especially when code needs to be included. This export process would need to be added as a project to their Quarterly Business Review (QBR) and be given priority to have a chance to be realized, as their team have their backlogs full for more than half a year at the time of the export request, with tasks of the new Azure pipeline and other Fortify tasks (upgrading and transitioning to Checkmarx). Also, extracting data with an SQL export would probably be without the code because the code might be in their file system and not in their database. This has not been verified as this SQL export option wasn't feasible in the first place.

When using the RestAPI of Fortify, .fpr files (actually a renamed .zip file) can be generated (contains all findings, tags, comments, relevant code etc.), downloaded and unzipped. The needed data can then be extracted by using a self-written parser to parse the XML files included in the .fpr file. Using this option, there is a choice to include or not to include the corresponding code in the compressed archive file, which may be useful at first to extract without code to save time and disk space. The code can be downloaded after manually choosing which project and its vulnerabilities have enough quality to be a part of the data set. According to the Fortify administrators, generating these .fpr files is also a heavy load for the Fortify servers, especially projects and are either sizeable or that are long-running or both. However, after these files are downloaded, parsing the XML files can be done locally and would not cause any load on the Fortify servers. Nevertheless, this generating and downloading process moments needs to be aligned with all users using Fortify. This is to reduce load on the server and prevent denial of service for other users, or to monitor the load if necessary.

A final option was thought by ourselves, that was possible without the help of any administrator, that is to scrape the data off the web app. The scraping of this website is more feasible than the first two options. A data set can be constructed with the data seen in figure 3.4. This is less data than what is provided in the .fpr files and thus such request made to the servers takes less load. There exist a browser automation tool called Selenium WebDriver[2]. Selenium's primary purpose is to automate web applications for testing purposes, but it can also be used for browsing websites and extracting HTML elements, which fits the goal of data extraction. Selenium available to multiple programming languages as a library. Working with Selenium as a Python library was already familiar, so the choice to use Python is quickly made. Usage of Selenium is quite simple. By attaching a web driver, such as ChromeDriver (Chromium browser) or FirefoxDriver (Firefox browser) to the Selenium object, Selenium can make use of these actual web browser for automation. Subsequently, Selenium can be made navigating to a given URL in these web drivers. Here on, to navigate through the website, Selenium uses HTML id's, class, tags etc. to find web elements on the website showing on the web browser. These can be buttons (e.g. next page button) or links (e.g. also a next page button) that can be clicked on by Selenium. Finally, after navigating to the string that needs to be scraped, Selenium can be programmed to copy strings in a certain HTML tag of the web page, so the task is to find the HTML tag in which the string is contained in.

3.1.4. Files inspection

After extracting the code, an inspection of the code is done to see what is in our code base. For the biggest part of the extracted code base, the files are in .java. However, there are also non-Java files, for example .properties,

[2] `https://www.seleniumhq.org/`

.xsd (XML Schema file), .wsdl (Web Service Description Language), .cs (C# programming language) and .scala files.

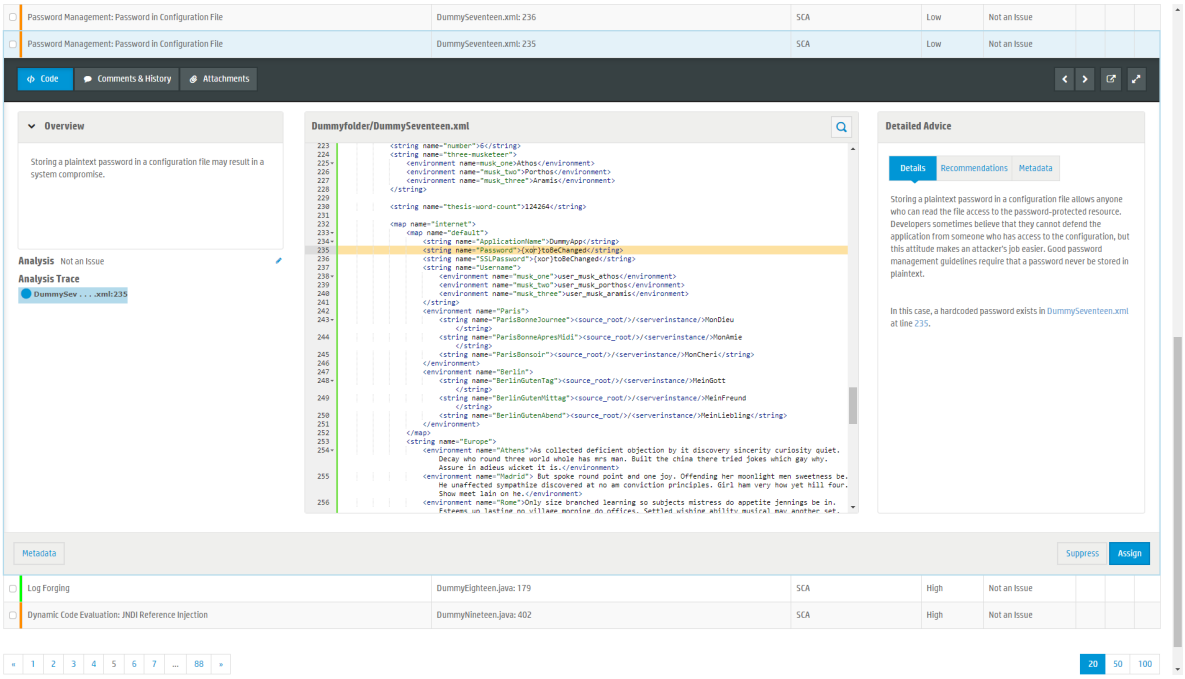


Figure 3.4: Details of the vulnerability, recommendations for fix and corresponding file and line number and shown when clicked on a vulnerability. With dummy names and numbers.

The files are diverse in the entire data set. There are the XML parsers, exception files and files that implement the factory method pattern, but also loggers, decorator, configurators, schedulers, filters, readers, data senders, adapters, handlers and the list goes on. The code is used for different settings. Some are used as a back-end for a mobile app, while others are being used as an API etc.

Files also vary in size/lines of code. Some files have hundreds of lines while others have 30 lines. Some files have tens of imports and methods while others have one import and three empty methods. This shows that there is a huge variety of code files, which is positive for our data set and hence our prediction model.

What is remarkable is that the exact same files appear in multiple projects. These files are libraries such as self-written parsers that is used across multiple projects. Not only do they appear in different projects, they also appear in the same project with just a different version numbers. The latter projects of course scan the same file as the same library is used.

Some projects have more vulnerabilities and more higher risk than others. Projects with the most critical vulnerabilities were not fixed as they were being phased out according to the developer teams of ING.

3.2. Data set creation, cleaning and analysis

Using Mauricio Aniche's open-source code metrics extraction tool for Java code with static code analysis [7], a class-level metrics data set containing was built using the data from Fortify. This tool is chosen over other tools because it is non-commercial, compatible with Java files, most complete and easy to use. It extracts the Chidamber and Kemerer objected oriented software metrics which is a popular and a successful set of metrics as explained in the literature. Also, some other software metric extraction tools needed the code to be compiled. This was a problem as we only have certain files of projects and therefore could not compile the code without the missing dependencies. Luckily, Mauricio Aniche's tool did not require the code to be compiled.

Our data set is comprised only of Fortify data. Checkmarx data did not have any useful data, as found potential vulnerabilities did not have a label validated by a pentester whether the potential vulnerability was truly vulnerable.

3.2.1. Data set cleaning and analysis

The class-level data set has 48 columns and 13647 rows. These exact columns are *project file class type cbo wmc dit rfc lcom totalMethods staticMethods publicMethods privateMethods protectedMethods defaultMethods abstractMethods finalMethods synchronizedMethods totalFields staticFields publicFields privateFields protectedFields defaultFields finalFields synchronizedFields nosi loc returnQty loopQty comparisonsQty tryCatchQty parenthesizedExpsQty stringLiteralsQty numbersQty assignmentsQty mathOperationsQty variablesQty maxNestedBlocks anonymousClassesQty subClassesQty lambdasQty uniqueWordsQty modifiers warningType, lineNum, nested, vulnerable* with the last columns as the target column.

3.2.2. Data set features explanation

There are 48 columns in our data set. All object oriented metrics that Chidamber and Kemerer proposed in their paper, as explained in the literature, appear in this feature set. Others features have names that speak for itself. Nevertheless, each of them will be explained.

`project` is the name of the project, as given in Fortify, that contained the potentially vulnerable file.

`file` is the complete path of the Java file.

`class` is the name of the class prepended by the name of the Java package that contains the file.

`type` is an enumeration of `class` when it is a normal class file, `interface` when it is an interface, `anonymous` when it is an anonymous class or `subclass` if the class extends another class.

`cbo` (Coupling between objects) counts the number of classes that the class is dependent on. This can be any type of dependencies, such as method return type, variable declaration etc. It does ignore dependencies to Java itself, like Java.lang.String.

`dit` (Depth Inheritance tree) is the number of parent node a class has, all the way up to Java.lang.Object. Every class has a `DIT` of at least one because in Java all classes are decending from Java.lang.Object.

`rfc` (Response for a Class) is the number of unique method invocation in a class i.e. it measures the number of distinct methods and constructors that can be executed

`wmc` (Weight Method Class) is the McCabe's complexity as explained in the literature. It counts the number of branch instructions in a class.

`totalMethods staticMethods publicMethods privateMethodsprotectedMethods defaultMethods abstractMethods finalMethods synchronizedMethods` denotes the amount of methods in total methods, static methods, public methods, private methods, protected methods, default methods (methods with no access level modifier), abstract methods, final methods and synchronized methods of a class respectively. Constructors are counted as methods too.

`totalFields staticFields publicFields privateFields protectedFields defaultFields finalFields synchronizedFields` denotes the amount of fields in total fields, static fields, public fields, private fields, protected fields, default fields (fields with no access level modifier), abstract fields, final fields and synchronized fields of a class respectively. A field is a class, interface or enum with an associated value.

`nosi` (Number of static invocation) is the number of static methods invoked by the class. Only the static methods that can be resolved by the JDT (Java development tools) are counted.

`loc` stands for lines of code, but this metric is actually the source lines of code. The difference is that in the source lines of code, empty lines and comments are not included.

`lcom` (Lack of Cohesion of Methods) The number of disjoint sets formed by the intersection of the *n* sets of instance variables used by all *n* methods of some class C.

`returnQty loopQty comparisonsQty tryCatchQty parenthesizedExpsQty stringLiteralsQty numbersQty assignmentsQty mathOperationsQty variablesQty maxNestedBlocks anonymousClassesQty subClassesQty lambdasQty` are the number of return statements, number of loops, number of comparison operators, number of try-catch statements, number of

expressions inside parenthesis, the number of strings, the number of numbers (int, long, double, float), number of math operations (times $*$, divide /, remainder %, plus +, minus −, left shift << and right shift >>), number of declared variables, highest number of blocks nested together, number of anonymous classes, number of lambda expressions and unique words.

`uniqueWordsQty` is number of unique words in the source code. It uses the entire body of the class and it basically is the number of words in a method/class after removing Java keywords. Names are split on capital letters (if they are in camel case) and underscores.

`modifiers` are the number of public/abstract/private/protective/native modifiers of a class or method.

`warningType` is the type of vulnerability that is given by Fortify. In the appendix, table A.2, you can find all the warning types and its number of occurences in the data set.

`lineNum` is the line number of the potentially vulnerable line according to Fortify.

`nested` is the the level of parentheses of the vulnerable line.

`vulnerable` is the label whether the class is vulnerable or not.

Most of these columns are extracted from the tool, while others were manually added. The columns that were not from the tool were lineNum, nested and vulnerable.

3.2.3. Data set cleaning

In order to do classification efficiently, the data set requires cleaning. This is to reduce problems such as overfitting due to for example duplicates and incorrect generalization by the machine learning algorithm because of null values or impossible negative values.

Some columns had to be removed immediately. These are `project`, `file`, `class`. These three columns are names and names cannot and should not be used as a feature to predict whether code is vulnerable or not. `linenum` is also removed, as, in our opinion, from a line number you cannot deduct whether code is vulnerable.

At this point, a histogram is created for every column to show the distribution of values in the column. See figure 3.5 for a complete overview. The majority of the columns shows that almost all values are 0. We considered either keeping the values like this, or transform each value of those column to a boolean value, indicating whether the value was not 0 e.g. 1 for everything greater than 0 and 0 if the value was 0. It was decided to keep the values as they are, as transforming might change the eventual classification negatively.

When checking the standard deviation of every column of the data set, this column returns a 0. Which means all values in this column as the same, meaning it does not give any information. The values are inspected and they are all zero. Apparently, there were no synchronized fields found in the code base. Therefore, `synchronizedFields` is removed.

To analyze whether the columns are correlated, a heatmap is produced using Pearson's correlation matrix. See figure 3.6. It can be seen that from the coloring that there are some highly correlated columns. For example `loc` and `variablesQty`. This is can logical as more code means more chance there is a vulnerabilitys. Also, none of the individual columns in the data set have a correlation with the target column. So, the software vulnerability detection problem on this particular data set is non-trivial. To give a better overview of how many columns are correlated, the numbers of the heatmap are counted and plotted in a bar chart. See figure 3.7. Each bar represents a column-pair.

Null values exists in columns `vulnerabilityType`, `lineNum` and `nested`. When one of these columns have a null value, the values of the other two columns also have null values. The null values exists because Fortify sometimes did not show a line number. Therefore, nested could not be calculated. The amount of null value rows is 252. As these null value rows are a small portion of the entire data set and null values cannot be filled, these rows are simply removed.

Each columns are checked whether it contained negative values, and indeed it does. Negative values are only found in `maxNestedBlock`, `subClassesQty` and `modifiers`. This might occur only when the `type`

value is anonymous. All these negative values are all -1, indicating that the value in the column is not applicable. These 1328 rows are removed.

In the previous section, we mentioned that there are duplicates in our code base. We found out that two thirds of the left over data set are duplicates. Some files are used in multiple projects as a library and are referenced in every project that was scanned. Also, some files have multiple potential vulnerabilities. This also creates duplicates. Our first intuition regarding duplicates is to remove them. The k-means algorithm uses the mean to calculate centroids. By having duplicates, the mean will change to and the algorithm will be biased. 8298 of 12067 rows are duplicate, leaving 3769 rows. 459 of these 3769 can be counted as duplicate rows if we leave out the `vulnerable` column. These 459 rows have at least one `TRUE` label and at least one `FALSE` in the data set, of which 244 are `TRUE` labels and 215 are `FALSE` labels. This is possible because for the same file that were scanned in Fortify, different lines were marked as potential vulnerabilities. While the 8298 rows were removed, the 459 rows were kept that way, because this rows reflects the real world.

3.2.4. Summary

Fortify is a static code analyzer deployed at ING. It uses a knowledge base to check whether code satisfy a rule that indicates a vulnerability. Servers running this static code analysis tool keep records of all software vulnerabilities found in the past with their corresponding code and level of risk. This is useful data concerning this research. After extracting the code, its vulnerability label and other data from Fortify by scraping the web app, a data set is created using an open-source tool by Mauricio Aniche. This tool calculates the class-level metrics in Java projects using static code analysis. The data set started with 48 columns and 13647 rows. In total have we removed 5 columns. We have removed 3 columns `project`, `file` and `class` as they are just names. Another column `synchronizedFields` was removed as it contained only zeroes. The final column to be remove is `linenum`, as this indicates on what line number the vulnerable line is located at and it probably is random. For the rows, 252 null values were removed. Another 1328 rows with negatives values were deleted. Finally, another 8298 duplicate rows were taken out. This leaves us with a data set of 43 columns and 3769 rows. We make a distinction between original data set and cleaned data set throughout the rest of this report.

It is important to note that none individual column in the data set is highly correlated with the `vulnerable` column. It means that the problem we are addressing is not a trivial one with respect to this data set. Also, some columns are (highly) correlated. This will be dealt with in the next chapter by applying PCA.

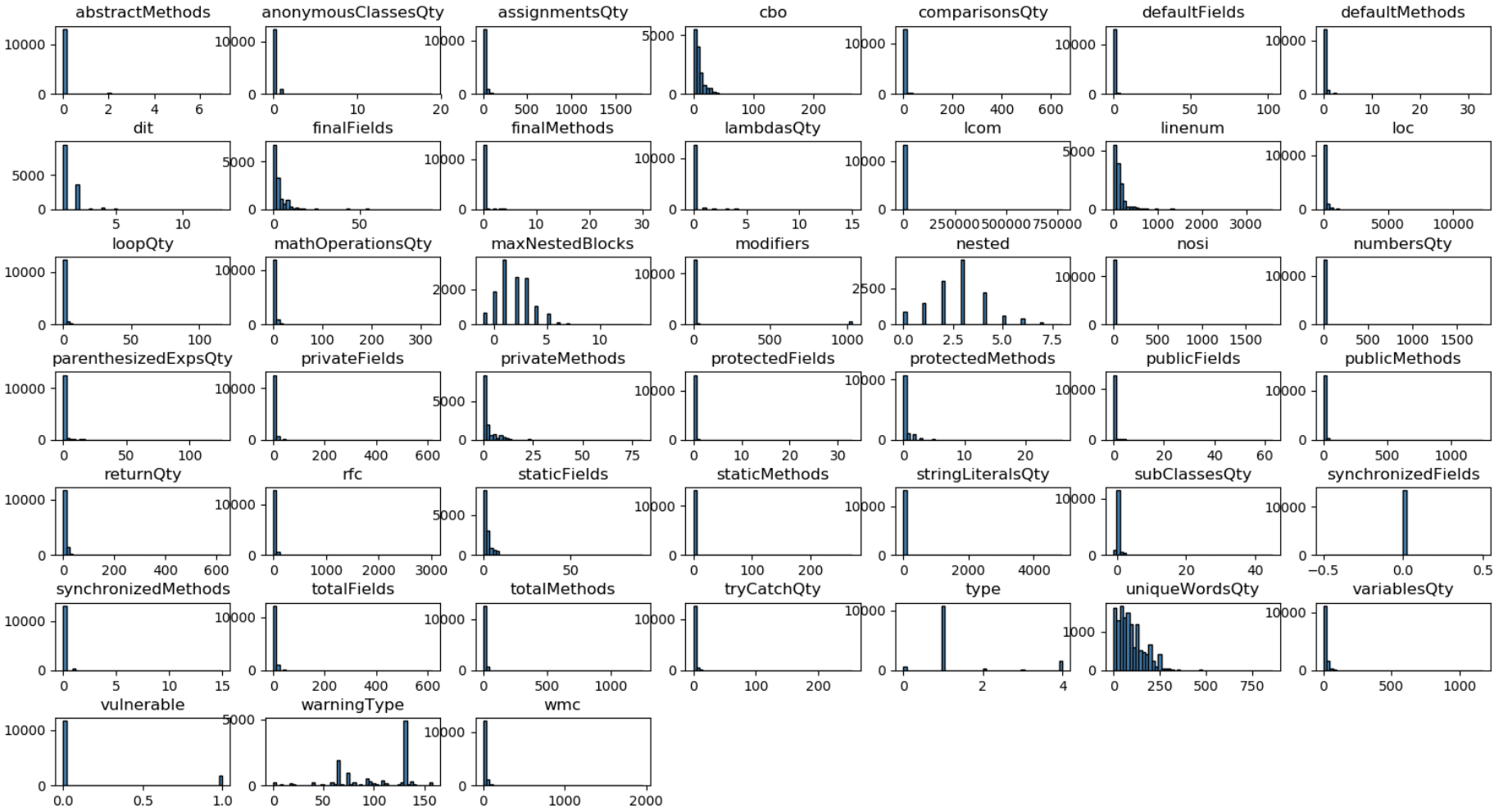


Figure 3.5: Histogram overview of columns. X: value, Y: occurences

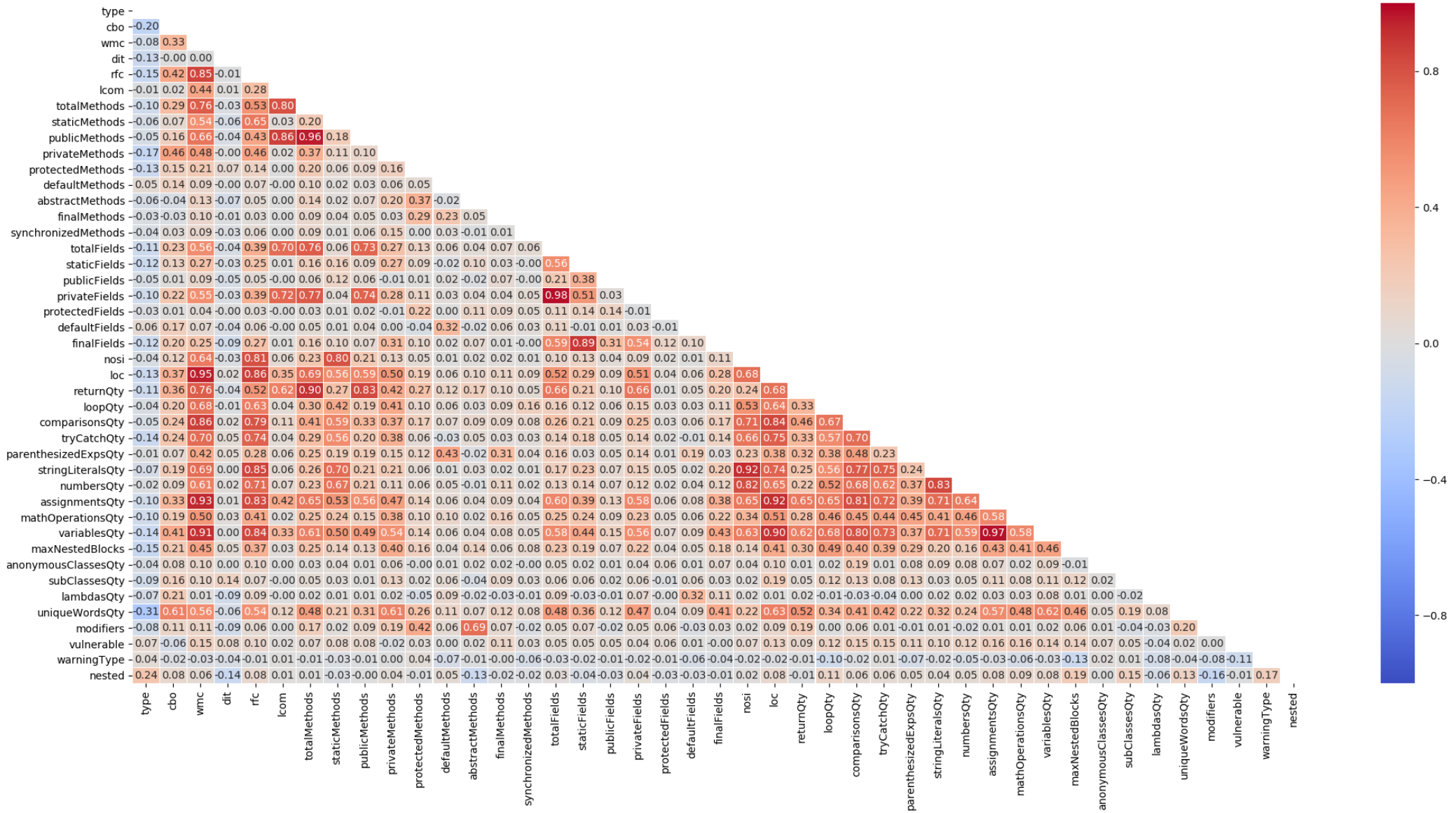


Figure 3.6: Heatmap from Pearson's correlation matrix

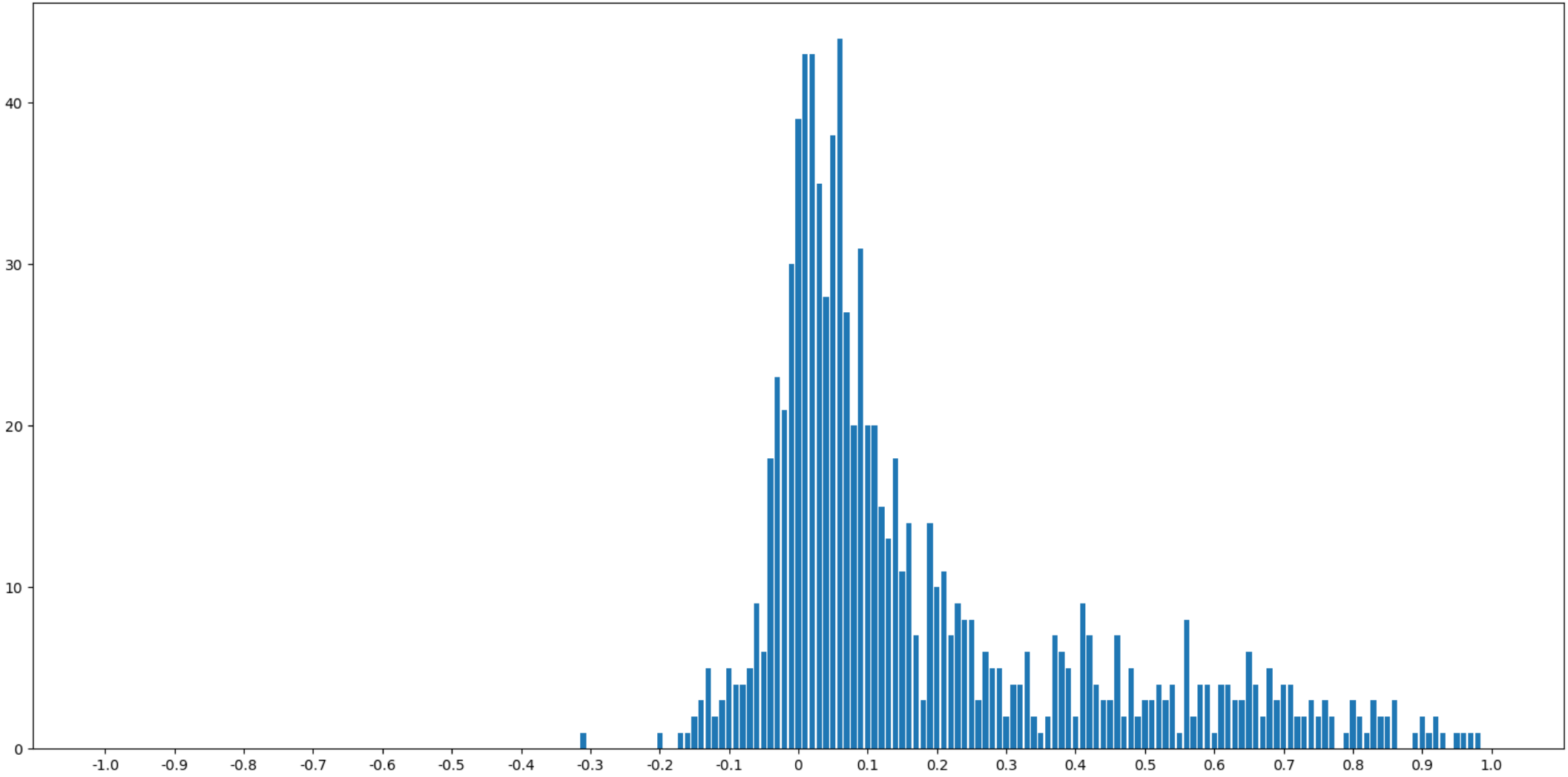


Figure 3.7: Correlation barchart. Each bar represents a column-pair

4

Clustering

Now that the data set has been analyzed. We can start building a prediction model. We can do pre-processing based on related work and the analysis of our data set.

4.1. Dimensionality Reduction using Principal Component Analysis

As the columns are correlated, some dimensionality reduction technique should be applied. We have chosen to apply Principal Component Analysis (PCA) to our data set, as an advantage is that PCA also suppresses noise. In the related work is shown that this could produce better results in software vulnerability detection research. When applying PCA to our data set, the data needs to be scaled first. This scaling is necessary when applying PCA, because the magnitude of the data in the every column is different. For example, `LOC` is usually much higher than `DIT`. PCA reduces dimensionality but attempts to keep as much variance as possible to have minimal loss of information. A column having a higher magnitude results in a higher variance, thus it will give an unwanted behavior in which PCA will be more biased to the column with the higher magnitude i.e. a higher weight. The data x is scaled by removing the mean and scaling to unit variance: z = (x - u) / s, where u is the mean of the training samples, and s is the standard deviation of the training samples [2]. The scaled number means how far the original value is away from the mean of that column that is being scaled, in terms of the standard deviation of that column. All columns now have the same potential weight to PCA.

When choosing the number of principal components, a minimum variance of 95% has been selected as a parameter for PCA. 5% of the variance is left out to suppress the noise. The PCA algorithm produced 26 principal components out of the 43 columns of the cleaned data set. See figure 4.3 for the variance distribution and figure 4.1 for heatmap matrix of the principal components coefficients i.e. the correlation between the principal components and columns of the cleaned data set.

After the reducing the dimensions, the data set will be divided into a training set and a test set. The training set is used to train our prediction model while the test set is used to evaluate our prediction model. The division is applied in such a way that the training set only contains data points with `FALSE` labels. The reason for this is to have `TRUE`-labeled data appear as outliers in our model when predicting the labels of new data. This will be further explained in the outlier detection section. The test set contains data points with both `TRUE` and `FALSE` in the same ratio of the cleaned data set, which is approximately 43%.

4.2. Visualization using t-SNE

To visualize our PCA'ed data, we use t-SNE as visualization method. This visualization is created on the PCA'ed data, rather than the cleaned data set, to have the noise removed and it quickens the execution. The aim is to get insight on how the data points can be divided into groups based on their similarities, and therefore the

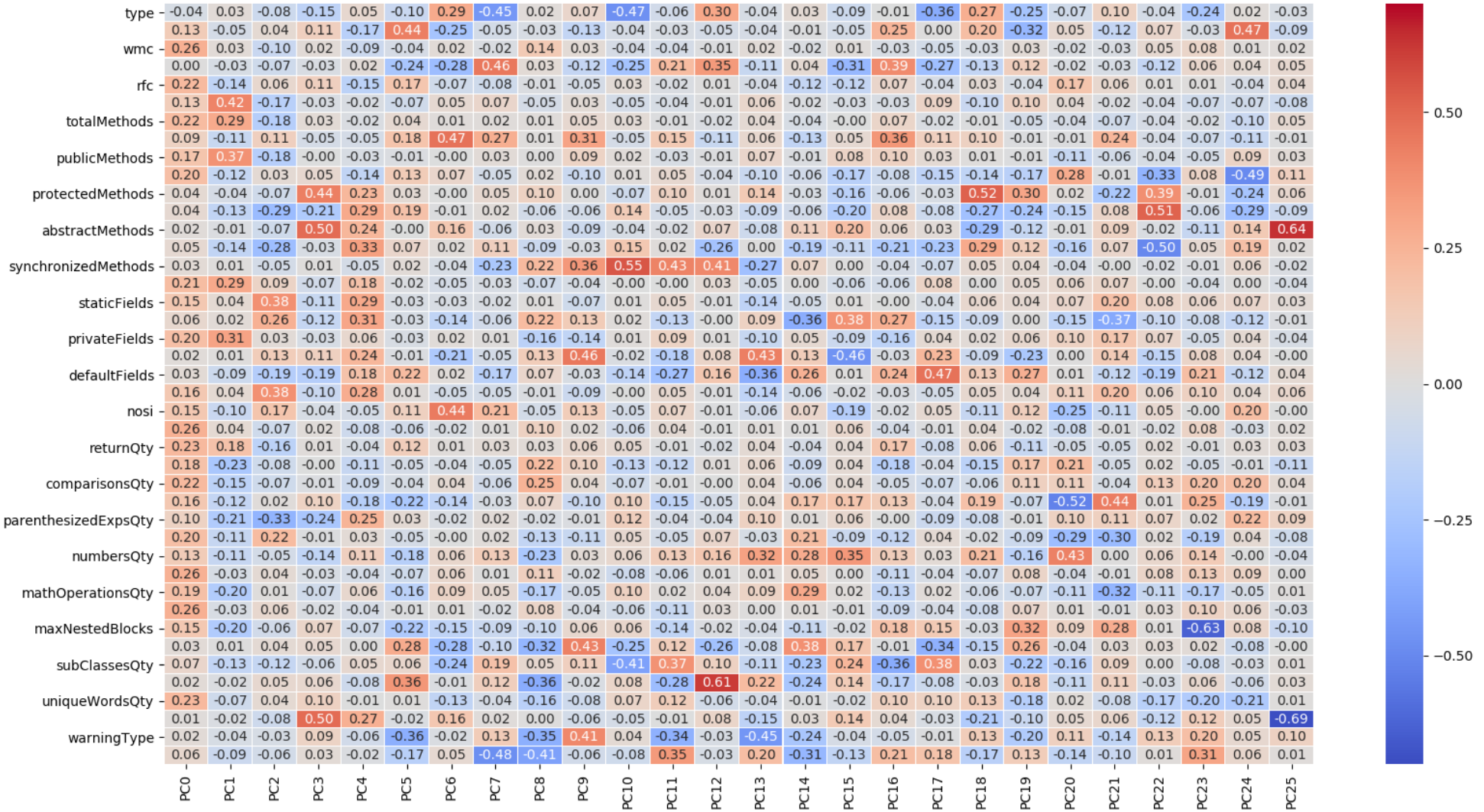
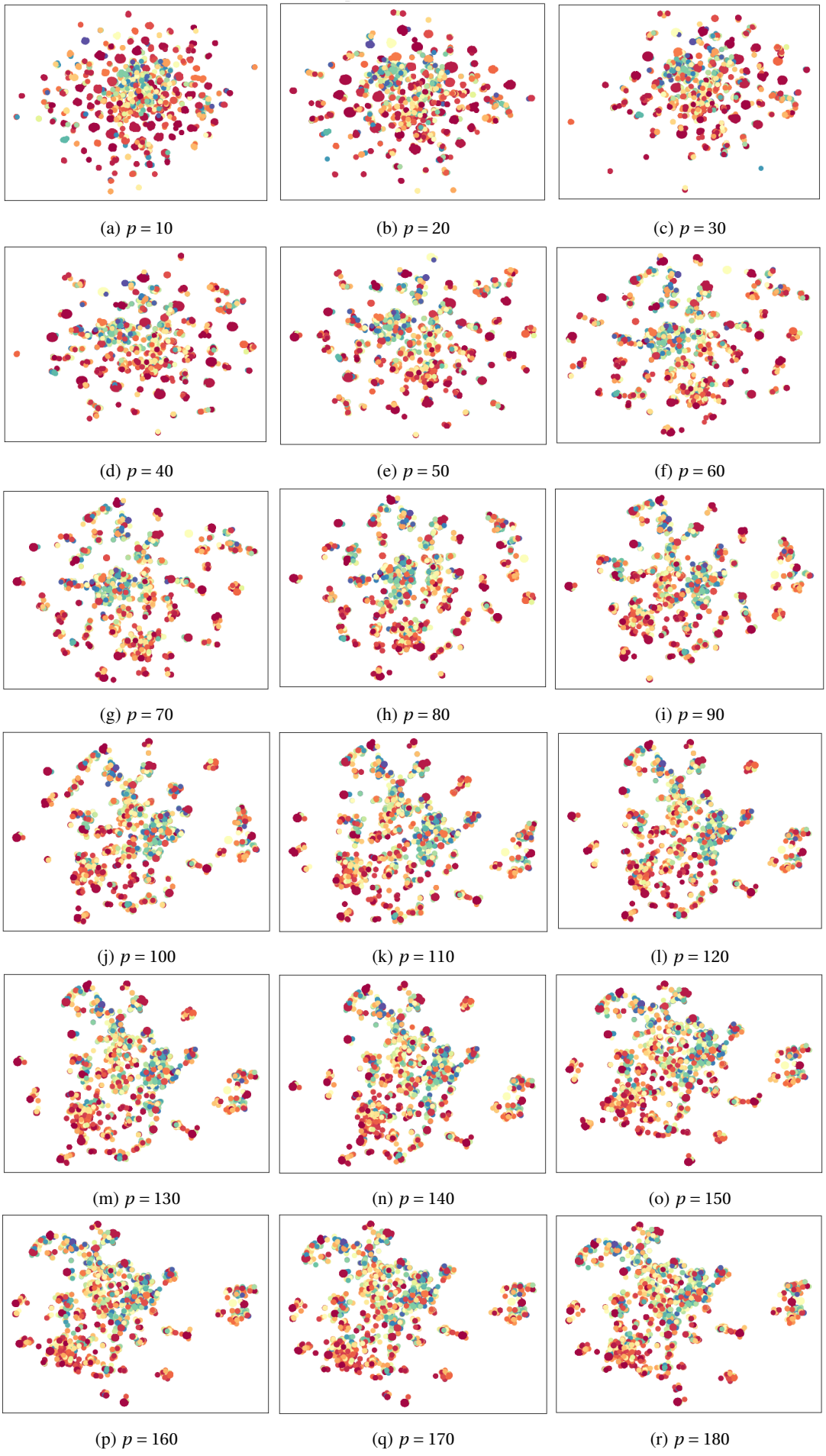


Figure 4.1: Heatmap on columns of the cleaned data set vs principal components



(a) $p = 10$ (b) $p = 20$ (c) $p = 30$

(d) $p = 40$ (e) $p = 50$ (f) $p = 60$

(g) $p = 70$ (h) $p = 80$ (i) $p = 90$

(j) $p = 100$ (k) $p = 110$ (l) $p = 120$

(m) $p = 130$ (n) $p = 140$ (o) $p = 150$

(p) $p = 160$ (q) $p = 170$ (r) $p = 180$

Figure 4.2: t-SNE visualizations

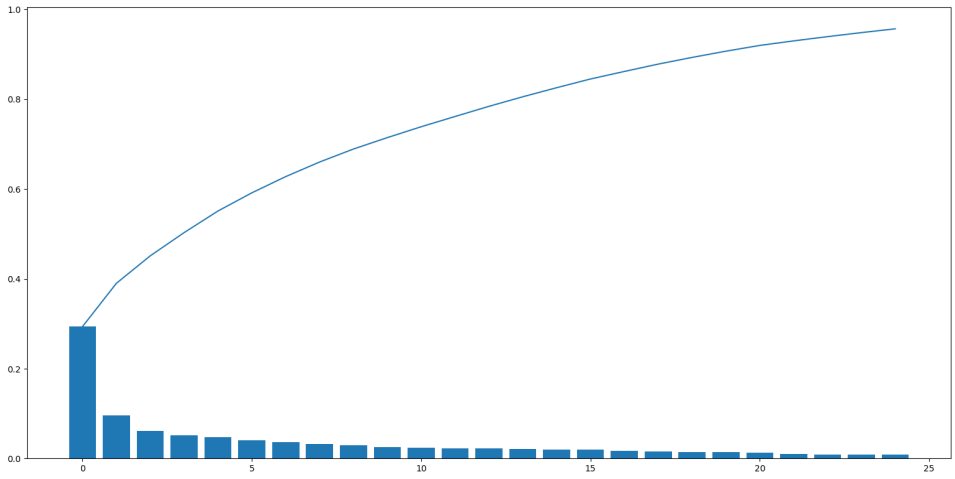


Figure 4.3: Variance in each principal components and its accumulation

number of clusters we should use when clustering with k-means. Multiple t-SNE visualizations are created with a different parameter perplexity. See 4.2. Unfortunately, this graph does not give us any helpful insight if and how the data points can be divided. Even increasing the perplexity even further, playing with other parameters such as number of iterations, does not change the graph.

4.3. Clustering process

In this subsection, the choice of the algorithm will be explained and the application of the chosen k-means algorithm will be described.

4.3.1. Choice of (unsupervised) algorithm

As stated before, the original research was split into two parts. Whereas the parallel study by Dinesh Bisesser deals with supervised learning algorithms, this research looks into unsupervised learning algorithms. So the choice of algorithms is narrowed down to only unsupervised algorithms.

A few algorithms were considered for this study. These algorithms were k-means, Agglomerative Hierchical clustering, DBSCAN and Local Outlier Factor (LOF). The first three algorithms are clustering algorithms and the last algorithm is a density-based algorithm. The idea of using Local Outlier Factor was dropped quickly, as it produced bad initial results. Even trying different parameters, the F1-score was a mere 0.40.

As previously mentioned in the literature, k-means is shown to work in conjunction with PCA in software vulnerability detection. This could not be said for the other clustering algorithms. As k-means already produced good clusters with PCA, as described down below, the other clustering algorithms were left out of this study.

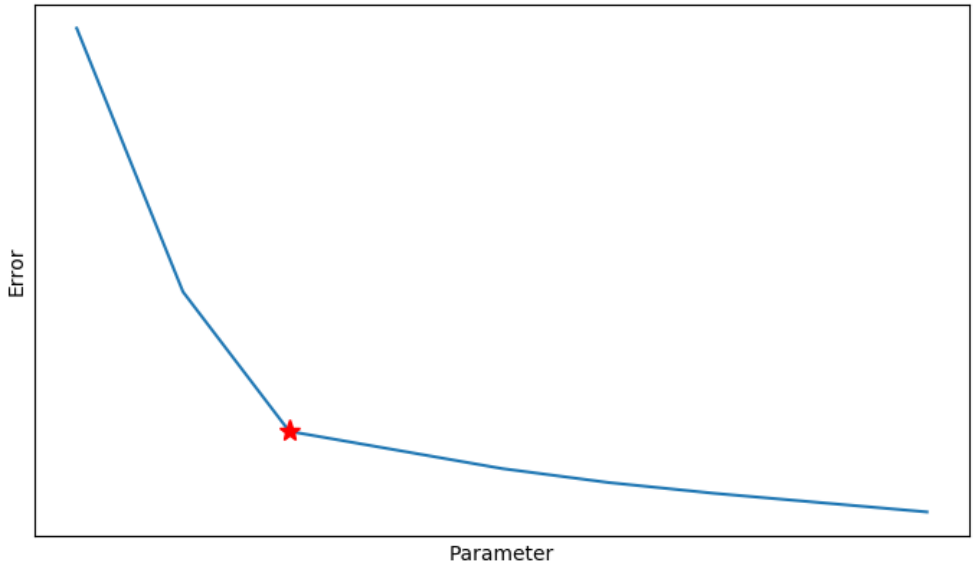
Also, other specific types of algorithms were also left out. For example, Neural Network was left out because it might be hard to train compared to the simpler algorithms mentioned above [18].

4.3.2. Application of the k-means algorithm

With a training set constructed, the prediction model can be created by initializing the k-means algorithm. But first, an appropriate k needs to be chosen as the k-means algorithm needs this parameter to initialize. The elbow method (see Intermezzo 1) is a good start for finding the proper k. This elbow method could normally also be used for choosing the number of principal components, However, in our case the elbow method is deemed a bad idea. In figure 4.3, there is not really an elbow to choose the number of principal components. Therefore,

Intermezzo: Elbow method

The elbow method is a technique to find an optimal parameter, applicable for algorithms such as k-means and PCA. This example illustrates a parameter vs error. If chart resembles an arm, then the point of inflection on the curve (the elbow) is a good indication of what parameter fits the model best. The idea is that by making the parameter larger, the trade-off with the error is not as big as before and therefore a larger parameter will not contribute more. In this case, the parameter at the red star is chosen.



Intermezzo 1: Elbow method

keeping an accumulation of 95% variance, while reducing 43 columns to 26 principal components, seems like a fair choice.

The idea of clustering is to group similar data points together. A trivial cluster example would be to cluster files with a high lines of code (e.g. 500 lines) together, separate from files with a low lines of code (e.g. 30 lines).

In an attempt to create logical and meaningful clusters, the cluster sizes should not vary extremely i.e. the smallest clusters having 1% of the data points whereas the largest cluster having 80%. The number of clusters should not be too large. The clusters should represent a certain or multiple characteristic(s), not represent individual data points. For our data set, choosing 1000 clusters would be too much, because some clusters would have 5 data points. A cluster with 5 data points does not represent a generalized and it is also unlikely that there are 1000 generalized characteristics. This kept in mind, the k-means algorithm is applied on our reduced data set for k ranging from 1 to 50. Due to non-deterministic nature of k-means, the algorithm is also applied 10 times. We use the average variance per k to determine what a good starting point is choosing the amount of clusters. The variance is to ensure that the sizes of the clusters do not vary to extreme as mentioned above.

Using the elbow method, a fair starting point of 7 clusters is chosen. See figure 4.4. A fixed seed to chosen for reproducibility means. To validate the meaning behind the clusters and whether this chosen clustering is logical, we manually inspect the data points within each clusters. Note that PCA was applied, so the data needs to be reverted back to the data of the cleaned data set. Luckily, data with PCA applied does not change its order, so reverting data back was just a matter of using the stored cleaned data set. Attaching the column of assigned cluster by the k-means algorithm, the data points can now be grouped to their respective assigned cluster.

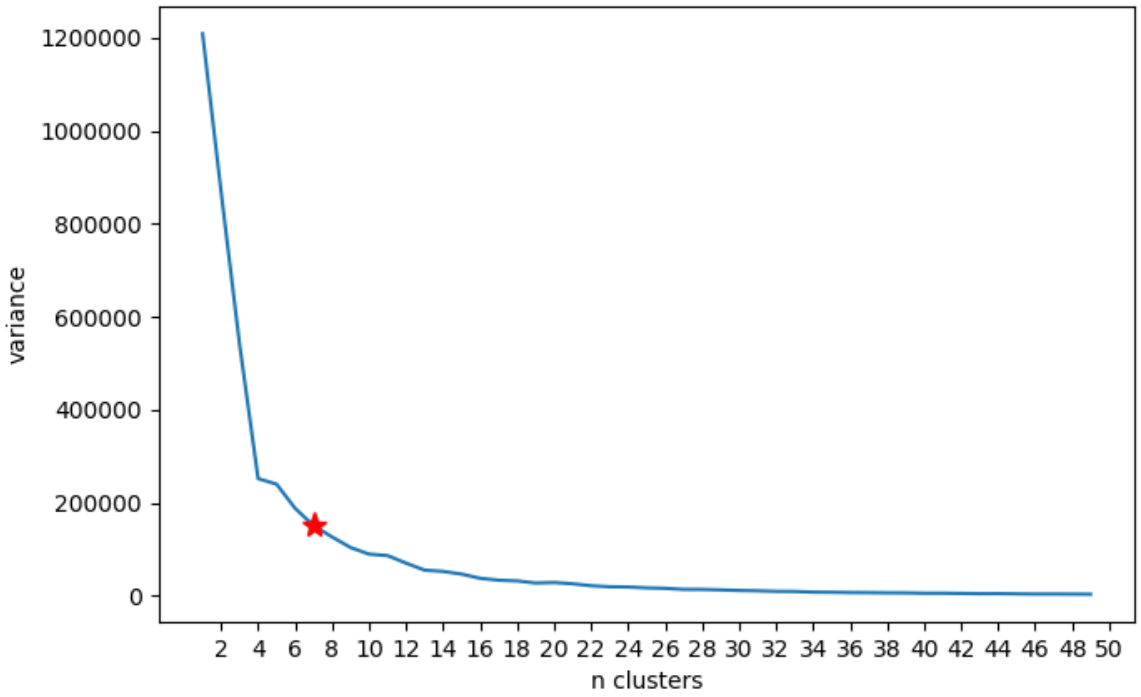


Figure 4.4: Variance vs cluster sizes. The red star represents the point chosen using the elbow method

4.3.3. Finding the reasoning behind clusters

Inspecting the clusters and finding out what the clusters mean would be easy by comparing the centroids for each clusters. However, the centroids retrieved from the *k*-means algorithm is in the form of principal components. From this data, it is unfortunately not possible to convert it to a representation using the original columns. Using basic linear algebra, the cleaned data set is multiplied with a transformation matrix to obtain the PCA data. This transformation matrix is, however, not invertible as it is not a square matrix. This would mean that the PCA centroids might not have a unique representation using the original columns. Solutions might be found using Matlab or any similar tool, but it would probably take too long as the transformation matrix has a size of 26 (number of principal components) by 43 (number of columns in cleaned data set).

Instead, to inspect the clusters, we tediously find the minimum value, the maximum value and the average value of each column in each clusters, then we compare these values for each cluster in order to find out the unique group of characteristics in terms of values of each columns i.e. what differs one clusters' values with another clusters' value. These minimum and maximum values per column per cluster can be found in table 4.1, 4.2, and the average values per column per cluster can be found in 4.3, 4.4. Please note that the original value and feature are shown in the tables, but the principal components and its corresponding values for each data point were used during clustering.

Clusters 1, 2, 3, 4, 5 were quite easy to determine its unique characteristics. See table 4.5. Cluster 1 had really low values. Most of them are 0 or some other values that is relatively low compared to all other values in the data set. for the same column. Cluster 2 has all (extremely) high values compared to all other clusters in the data set. Cluster 3 is empty for some unknown reason. Every time a clustering is performed with the *k*-means algorithm of the Python library *scikit-learn* on this data set, there seems to be always at least one cluster that is empty. One of a reason could be that some of the initialized centroids are initialized far from any other points and that these centroids converges. This is not a big issue and does not have any impact on the clustering, as our elbow method used is still valid, it is left this way. Cluster 4 have a high amount of methods and lines of code. Cluster 5 have a high amount of field and lines of code.

Cluster 0 and 6 were not so easy to determine its distinctive characteristics. It looks like that cluster 0 have average amount of methods, but most of the methods are private and protected methods. The number of private

		type	cbo	wmc	dit	rfc	lcom	totalMethods	staticMethods	publicMethods	privateMethods	protectedMethods	defaultMethods	abstractMethods	finalMethods	synchronizedMethods	totalFields	staticFields	publicFields	privateFields	protectedFields	defaultFields	finalFields
0	min	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	max	0	30	101	4	103	1096	49	9	45	14	15	5	6	13	1	50	15	10	50	10	3	14
1	min	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	max	3	39	35	5	41	433	35	9	35	9	8	9	0	10	2	26	19	19	26	11	15	19
2	min	0	3	30	1	42	0	13	0	4	1	0	0	0	0	0	16	3	0	6	0	0	3
	max	0	264	609	2	412	8060	141	33	137	80	14	3	0	3	15	93	91	53	74	33	14	92
3	min																						
	max																						
4	min	0	6	73	2	34	93	19	1	10	5	0	0	0	0	0	6	1	0	3	0	0	1
	max	0	11	94	5	44	351	35	18	14	9	0	15	0	9	0	28	28	7	21	0	5	28
5	min	0	3	19	1	17	0	5	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
	max	3	108	142	4	314	3344	96	57	90	31	12	11	1	10	5	66	61	25	66	29	6	54
6	min	0	0	5	1	5	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	max	3	61	91	13	122	1331	78	19	78	18	10	10	0	10	11	38	19	15	34	13	10	19

Table 4.1: Min and max values of clusters (1/2)

		nosi	loc	returnQty	loopQty	comparisonsQty	tryCatchQty	parenthesizedExpsQty	stringLiteralsQty	numbersQty	assignmentsQty	mathOperationsQty	variablesQty	maxNestedBlocks	anonymousClassesQty	subClassesQty	lambdasQty	uniqueWordsQty	modifiers	warningType	nested
0	min	0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	19	1	0	0
	max	30	466	26	4	25	10	4	61	16	83	24	51	7	2	2	3	361	1025	60	8
1	min	0	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0
	max	13	311	29	5	17	6	14	107	45	45	15	33	5	2	7	5	183	1026	63	8
2	min	3	234	9	0	0	1	0	15	0	23	1	52	1	0	0	0	231	1	0	0
	max	352	2955	139	47	182	84	34	440	69	602	113	382	14	5	10	0	538	17	49	4
3	min																				
	max																				
4	min	1	270	27	6	23	0	66	15	32	56	22	41	5	0	1	0	127	16	8	2
	max	15	1041	34	9	32	26	96	29	476	152	81	89	5	0	2	0	530	17	24	5
5	min	0	158	0	0	0	0	0	1	0	18	0	12	0	0	0	0	79	1	0	1
	max	102	1079	80	20	97	42	42	210	168	177	82	145	9	12	4	15	559	1025	62	8
6	min	0	32	0	0	0	0	0	0	0	0	0	2	1	0	0	0	25	0	0	0
	max	29	612	35	14	27	30	42	136	93	86	37	64	9	4	7	14	295	25	63	8

Table 4.2: Min and max values of clusters (2/2)

	type	cbo	wmc	dit	rfc	lcom	totalMethods	staticMethods	publicMethods	privateMethods	protectedMethods	defaultMethods	abstractMethods	finalMethods	synchronizedMethods	totalFields
0	0.00	9.71	25.90	1.35	26.39	95.49	12.35	0.88	5.66	2.82	3.71	0.17	1.07	1.39	0.05	6.10
1	0.37	6.60	6.65	1.44	9.63	6.19	3.69	0.42	2.94	0.48	0.19	0.09	0.00	0.06	0.01	2.40
2	0.00	54.57	251.57	1.14	139.50	1825.79	72.93	8.07	53.79	17.36	1.43	0.36	0.00	0.43	1.21	42.71
3																
4	0.00	9.23	82.54	2.23	42.46	238.38	32.15	2.31	11.62	7.77	0.00	12.77	0.00	7.69	0.00	9.00
5	0.10	19.36	69.91	1.46	77.33	207.91	23.47	3.79	13.32	8.99	0.85	0.32	0.01	0.28	0.09	17.03
6	0.11	12.63	25.92	1.37	34.18	39.09	10.31	1.24	6.39	3.34	0.37	0.22	0.00	0.12	0.11	6.79
all	0.26	9.72	19.61	1.42	23.31	47.10	8.00	0.98	5.18	2.17	0.43	0.23	0.04	0.19	0.05	5.17

	staticFields	publicFields	privateFields	protectedFields	defaultFields	finalFields	publicFields	privateFields	protectedFields	defaultFields	finalFields	nosi	loc	returnQty	loopQty	comparisonsQty
0	2.56	0.29	5.00	0.77	0.04	2.80	0.29	5.00	0.77	0.04	2.80	2.99	149.82	7.88	0.83	4.51
1	0.94	0.14	2.06	0.11	0.10	1.28	0.14	2.06	0.11	0.10	1.28	0.94	42.57	2.57	0.24	0.97
2	26.79	9.00	29.00	2.36	2.36	28.14	9.00	29.00	2.36	2.36	28.14	50.00	1394.50	55.86	13.86	70.50
3																
4	3.08	0.54	4.38	0.00	4.08	3.08	0.54	4.38	0.00	4.08	3.08	4.31	406.85	28.77	7.23	26.54
5	11.17	1.51	15.01	0.38	0.13	11.46	1.51	15.01	0.38	0.13	11.46	10.87	413.29	19.10	3.64	17.74
6	2.98	0.36	6.01	0.22	0.20	3.79	0.36	6.01	0.22	0.20	3.79	4.38	159.82	7.75	1.27	5.10
all	2.51	0.37	4.44	0.20	0.17	2.98	0.37	4.44	0.20	0.17	2.98	3.07	118.52	5.98	0.94	4.12

Table 4.3: Average values of clusters (1/2)

	tryCatchQty	parenthesizedExpsQty	stringLiteralsQty	numbersQty	assignmentsQty	mathOperationsQty	variablesQty	maxNestedBlocks	anonymousClassesQty	subClassesQty	lambdasQty	uniqueWordsQty	modifiers	warningType	nested
0	2.39	0.48	12.85	2.51	18.62	3.71	15.70	2.37	0.06	0.12	0.10	117.93	1000.01	25.52	2.54
1	0.68	0.30	4.91	1.05	5.93	0.75	5.45	1.22	0.03	0.05	0.08	43.82	5.47	16.90	2.01
2	15.21	12.64	118.43	27.14	271.86	37.00	180.79	5.79	0.36	1.79	0.00	351.86	4.43	13.07	2.14
3															
4	2.00	81.77	17.92	71.08	72.38	32.69	52.46	5.00	0.00	1.54	0.00	167.77	16.08	10.54	3.62
5	6.23	4.46	64.78	19.98	74.65	16.41	58.68	3.85	0.32	0.26	0.84	211.28	29.83	20.56	2.98
6	2.93	1.44	18.44	4.65	25.78	4.46	21.41	2.93	0.10	0.16	0.33	120.50	2.96	21.66	3.01
all	1.90	1.52	14.25	4.11	19.27	3.50	15.73	2.01	0.08	0.12	0.20	83.78	45.24	18.82	2.40

Table 4.4: Average values of clusters (2/2)

Cluster	#points	Characteristics
0	82	Average amount of methods with higher than avg private and protected methods
1	1232	Low values
2	14	Extremely high values
3	0	Empty cluster
4	13	High amount of methods and lines of code
5	149	High amount of fields and lines of code
6	617	Values close to overall average

Table 4.5: Clusters and its characteristics.

and protected methods are higher than the overall average of these two columns. Cluster 6 have mostly values that are close to the overall average.

To have an even better understanding of the clusters, the data points of every cluster is visualized. For each cluster, a parallel coordinates graph (see Intermezzo 2) is plotted. See figure 4.5-4.10. Cluster 3 is empty so its parallel coordinates graph is left out. To have a good overview in the graphs, all data points are standardized, as we also did before applying PCA.

Cluster 2 has lots of high values. Cluster 4 has a few data points and for every column the values are almost the same except for one data point, which seems to be an outlier to this cluster (and therefore an outlier to all other clusters as this data point is the closest to the centroid of cluster 4). For cluster 0, 1, 5 and 6, other than the few columns named that distinct these clusters from the other clusters in table 4.5, its values are close to each other. In figure 4.12 and 4.11, the parallel coordinates are combined in one plot, with cluster 0 and cluster 1 switching from background/foreground. You can see that cluster 5's values are for most of the columns above cluster 6, and cluster 6's values are for most of the columns above cluster 0 and 1. This is rather confusing, giving the intuition that cluster 6's average standardized values are higher than 0 and 1.

This is not true. If we split the standardized data set in six subsets by clusters. We then take the average of each columns, done six times for each of the subsets. We end up have six rows and for every column a value indicating the average. Afterwards, we calculate the average for each row, doing this six times again for each of the subsets, ending with six numbers. See table 4.6. These numbers indicate how much per cluster on average the values are above or below the mean, expressed in terms of standard deviation of the entire set. With these number, you can conclude that on average the values of cluster 6 are higher than on average the values of cluster 1, but lower than on average the values of cluster 0. In other words, the clusters are based on how high the standardized values are, with the exceptions of peaks on some columns explained in table 4.5.

Cluster	average of average	normalized
0	0.323	0.163
1	-0.252	0
2	3.263	1
4	1.614	0.531
5	0.915	0.332
6	0.131	0.109

Table 4.6: How much on average are the values are above or below the mean, expressed in terms of standard deviation of the entire set, per cluster

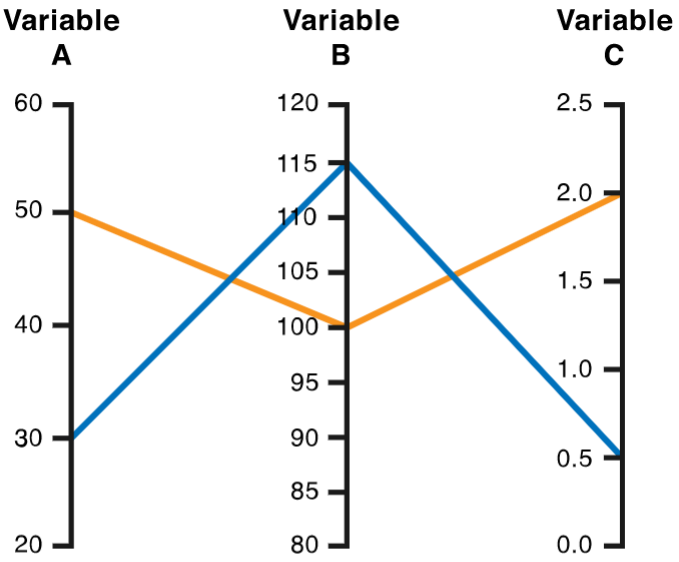
4.3.4. Files in each clusters

Each of these clusters are double-checked whether the files correspond with the metrics. A small sample of 2-3 files of each clusters were checked. And indeed, the code of all samples taken are expected as they should be. Files of cluster 2 were for example large and seemed complex and files of cluster 1 files with one or two methods that has a body of just one line, returning something.

It would be interesting to show what kind of files are in each clusters. This is however not possible to show, as it seem to be random. An example are multiple XML parsers found to be in clusters 0 and 6, different kind of exception files in clusters 0, 1, 5, 6, files that implement the factory method pattern in clusters 1, 5, 6.

Intermezzo: Parallel coordinates plot

The parallel coordinate plot is a visualization method used for plotting multivariate, numerical data [1]. Each data point is represented by a line, that goes through different axes that in turn represent each feature of the data set. The different axes of each feature may have its own scale and the line goes through these axes at the point of the value for that feature in the data point.



Data			
	Variable A	Variable B	Variable C
Item 1	50	100	2.0
Item 2	30	115	0.5

Intermezzo 2: Parallel coordinates plot

4.4. Summary

Starting off with a cleaned data set, we first scaled data set by standardization so that there won't be any bias when applying PCA. After that, we applied PCA to reduce the dimensions in our data set. PCA also reduced some noise when it leaved out 5% variance out of the data set. To visualize the data set, we used t-SNE and unfortunately, that did not give us the wanted insight on the reduced data set. Nevertheless, we went on to the clustering. As algorithms such as Local Outlier Factor did not suffice, the choice was to cluster using k-means. It clustered the data points in seven clusters (one empty). To find the logical reason behind the clusters, we looked at the minimum value, maximum value and average value for each columns of each clusters and deduced the reasoning behind the clusters. We also used parallel coordinates plots to visualize the data points of each clusters. There are some overlap in the parallel coordinates. It was expected that the lines would be cluttered and not completely divided as the software vulnerability detection method is a hard problem. Even so, there are still some patterns to be seen in the plots, as there are some colors that stick above other colors. The files have been manually validated to show the same characteristics as the other files in the same clusters. We deem the data points are well-clustered enough, hereby ending this section of clustering of the training set. The next step is to assign data points in the the test set to the created clusters and test whether vulnerable data points can be predicted by using our distance-based anomaly detection method.

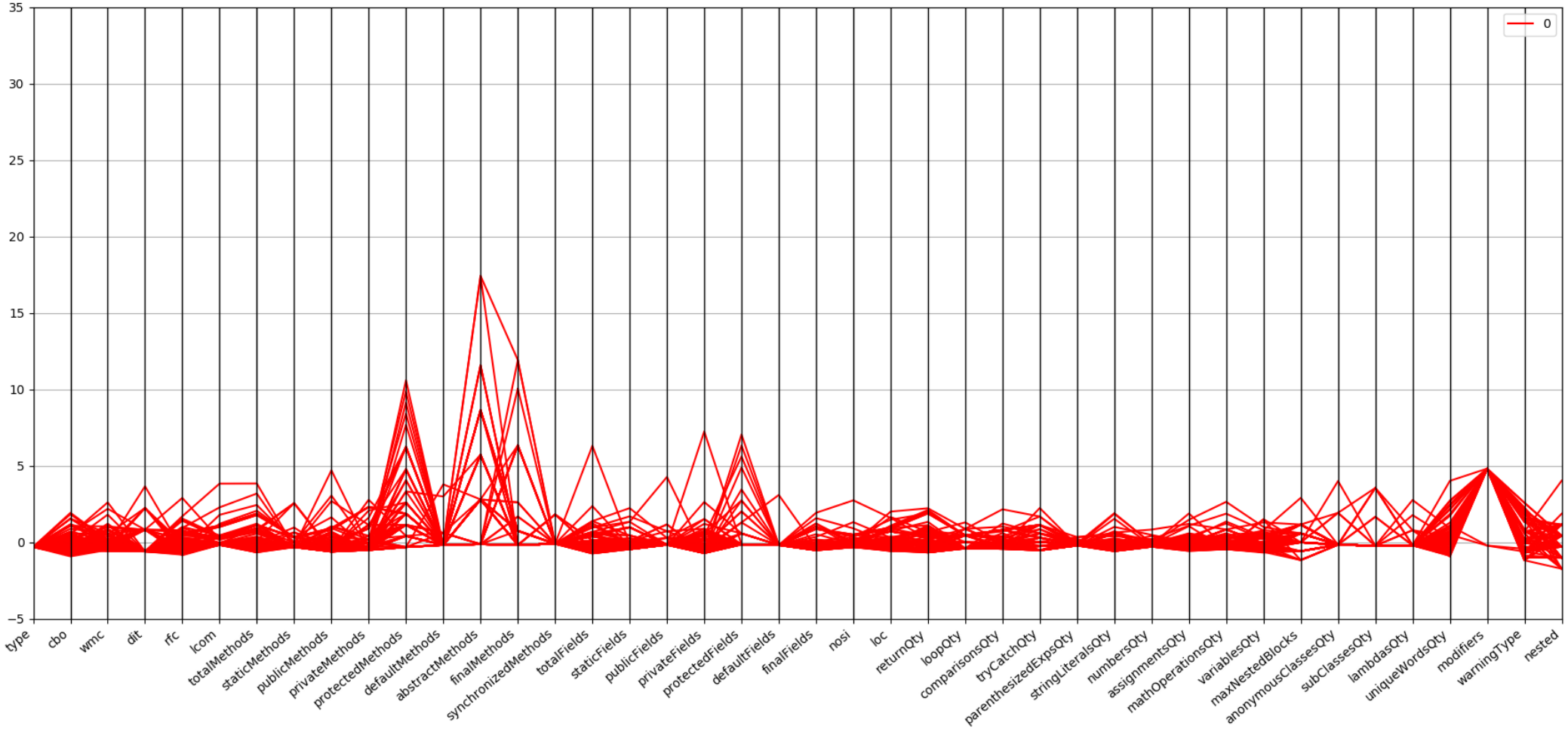


Figure 4.5: Parallel coordinates plot for cluster 0

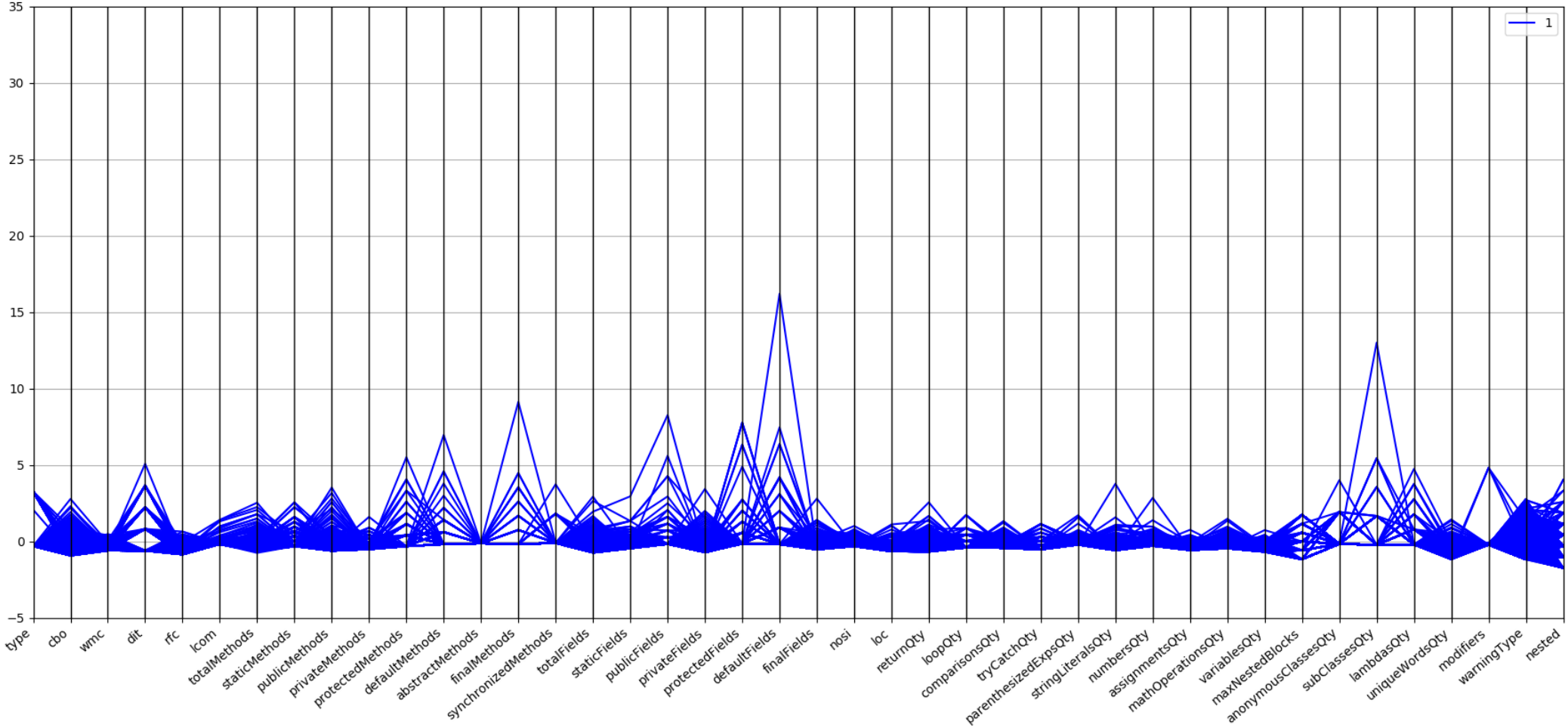


Figure 4.6: Parallel coordinates plot for cluster 1

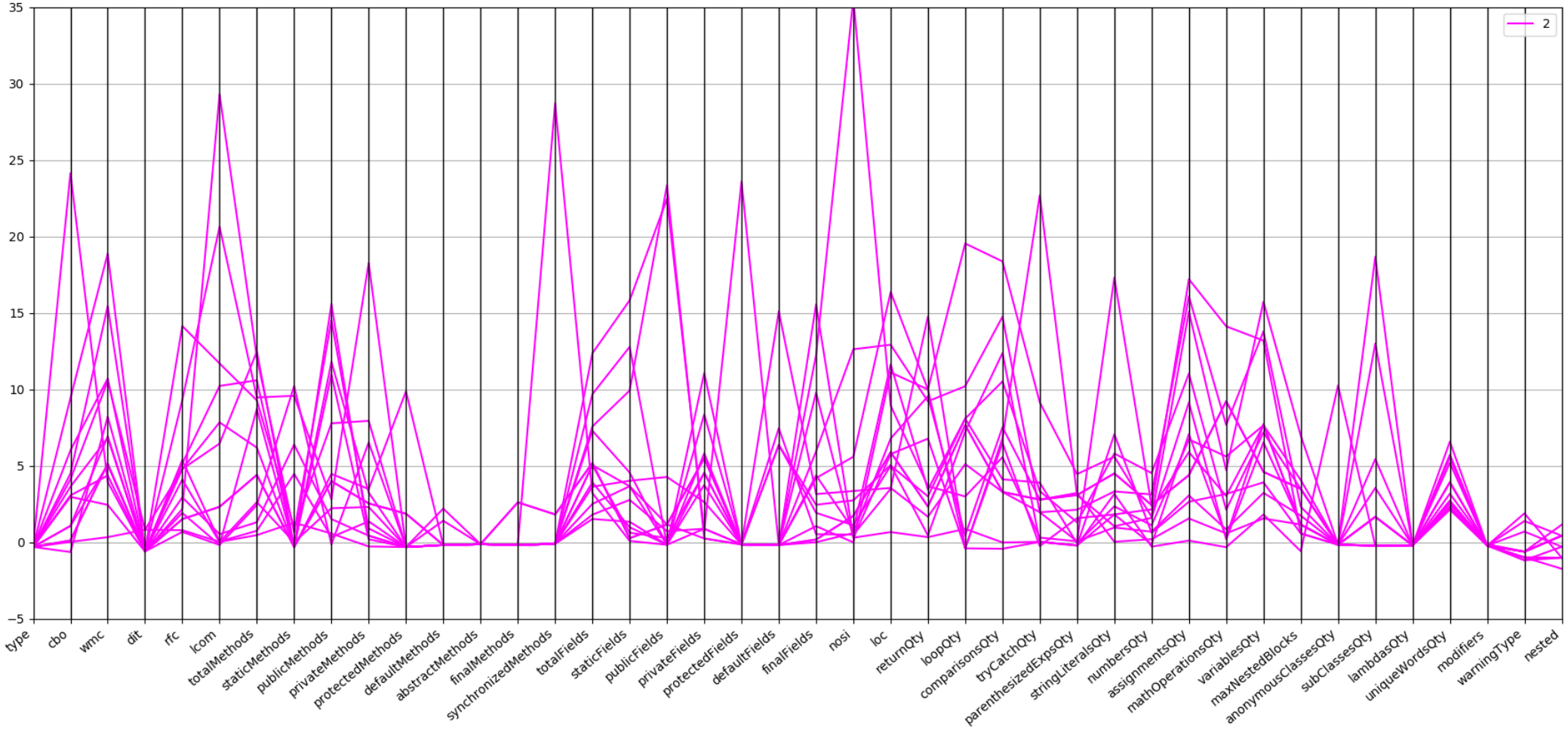


Figure 4.7: Parallel coordinates plot for cluster 2

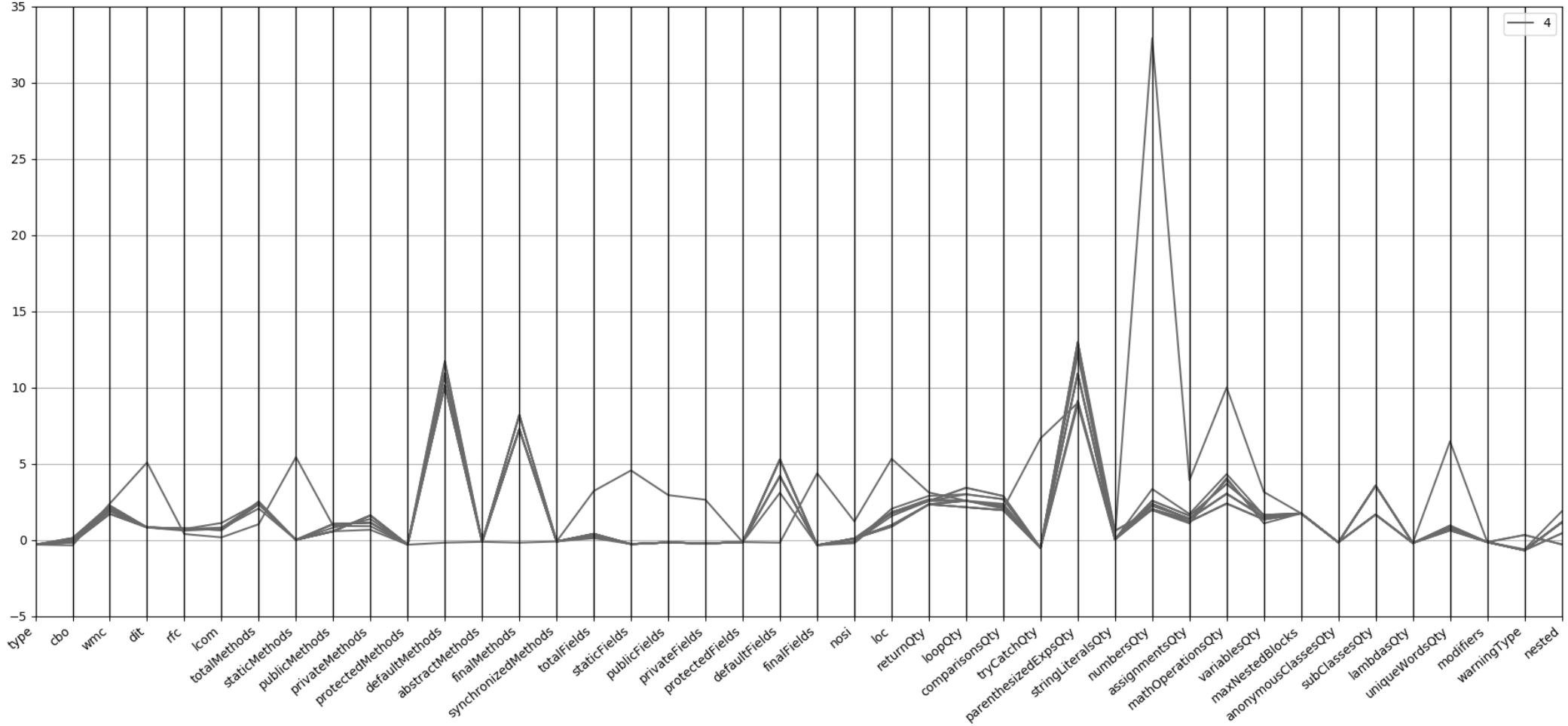


Figure 4.8: Parallel coordinates plot for cluster 4

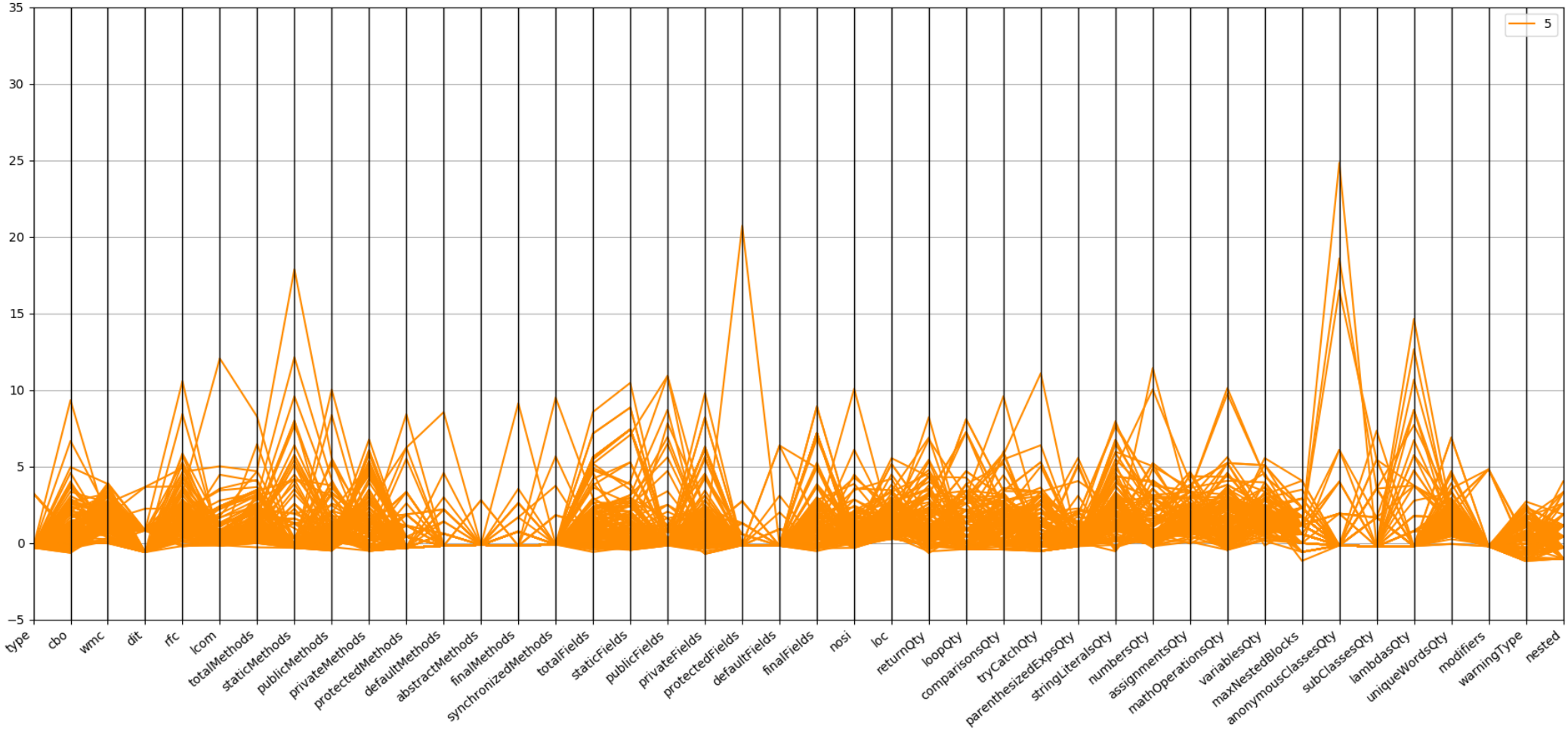


Figure 4.9: Parallel coordinates plot for cluster 5

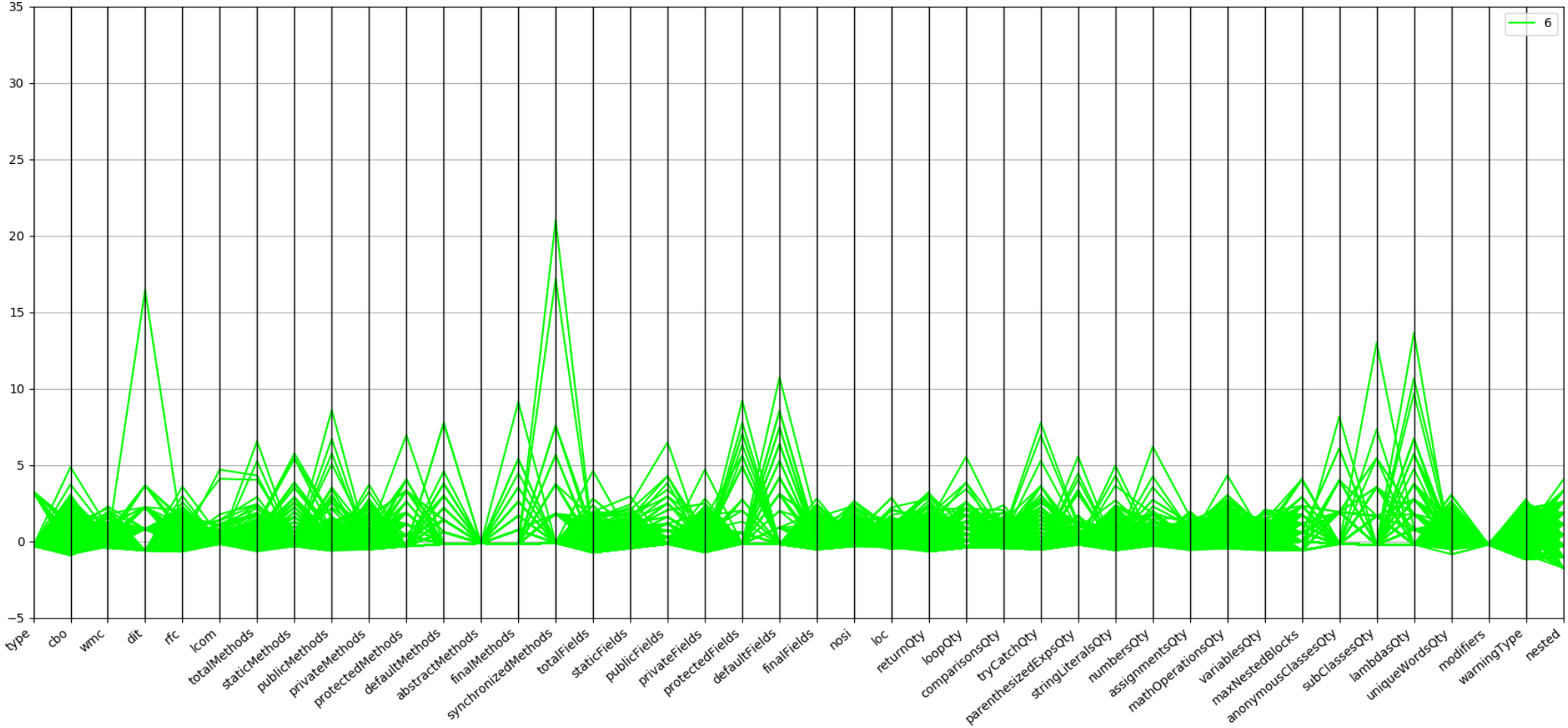


Figure 4.10: Parallel coordinates plot for cluster 6

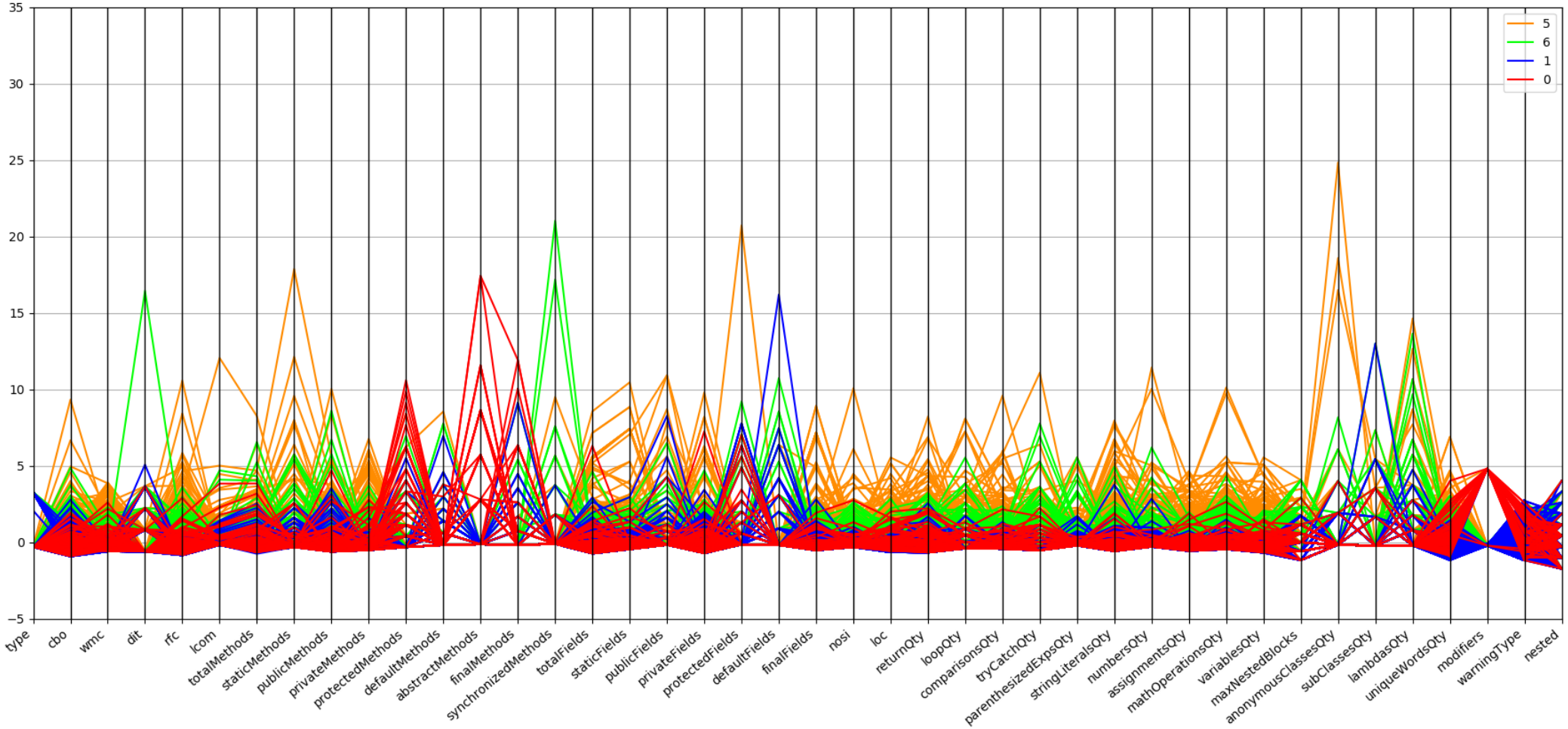


Figure 4.11: Parallel coordinates plot for cluster 0, 1, 5, 6 with cluster 0 on the foregrond

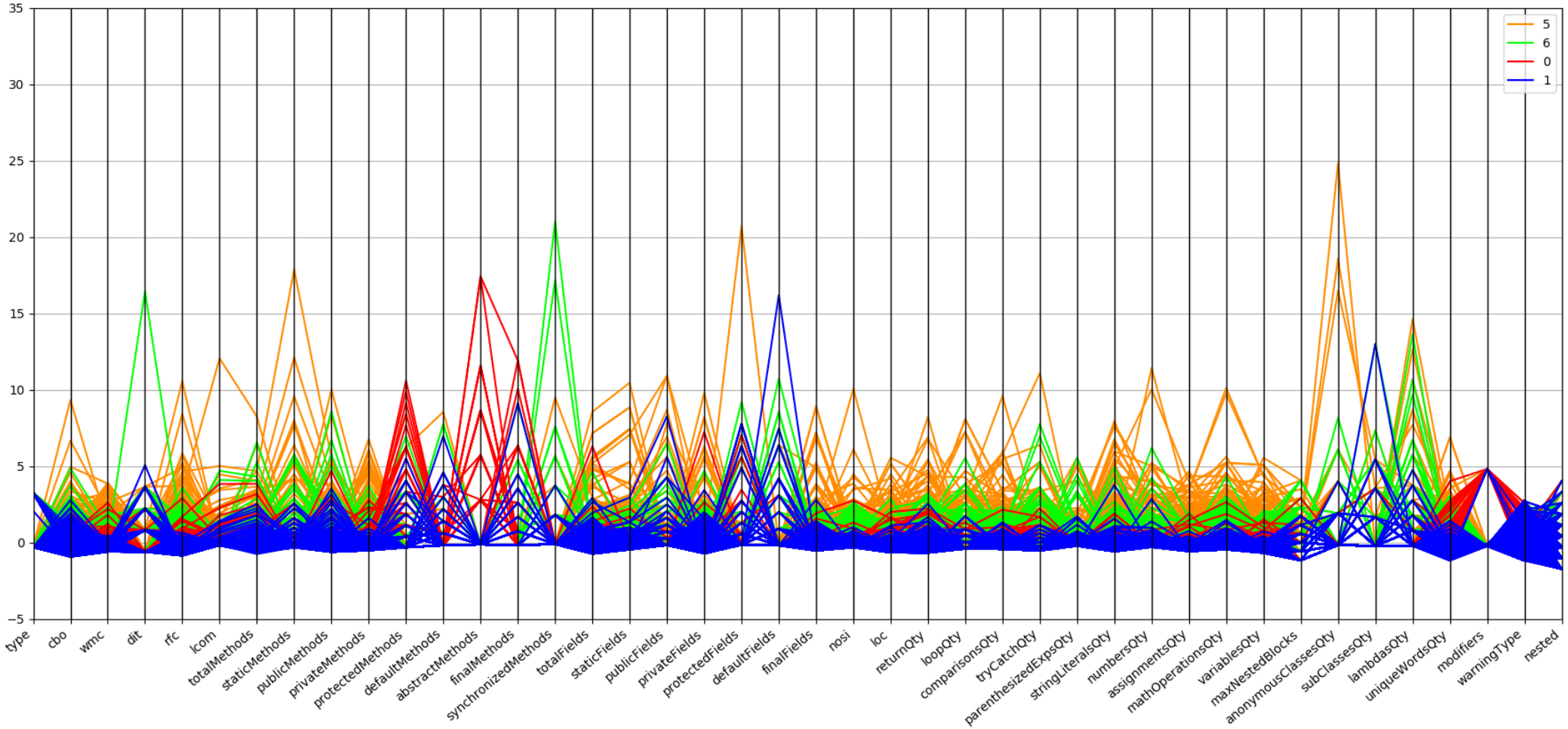


Figure 4.12: Parallel coordinates plot for cluster 0, 1, 5, 6 with cluster 1 on the foregrond

5

Anomaly detection

5.1. An outlier detection method

The assumption of the data set is that (1) different groups of `FALSE`-labeled data are similar to other `FALSE`-labeled data within the same group and (2) `TRUE`-labeled data are different from `FALSE`-labeled data. In other words, `FALSE`-labeled data should be clustered together and `TRUE`-labeled data should be outliers to those clusters.

If (1) would be false, having good quality clustering of these groups of non-vulnerable data points would not be possible. Supposing that (2) is false, then any detection method could not distinct vulnerable data points with non-vulnerable data points, rendering machine learning useless in this situation. Keep in mind that bad end results do not necessarily mean these assumptions do not hold.

With those assumptions kept in mind, predicting whether a new data point should be labeled as `TRUE` or `FALSE` is straight-forward. After training the model with the training set, the k-means algorithm has assigned all data points of the training set to a cluster. For each of these clusters, we calculate the distance from the centroid of the cluster to every point in the same cluster. Take the maximum distance of the calculated distances. This distances now defines the radius of that cluster. Every single cluster now has a radius.

5.2. Choice of the distance metric

The distances between data points play an important part in this outlier detection method. The choice for the distance metric is coherent with the curse of dimensionality problem, as the distance metric might bring the curse into play or even intensify the curse. As stated in the literature by Aggarwal et al., the Manhattan distance is a more meaningful distance metric than the Euclidean distance is the dimensions are too high. To test whether the distances make any differences in terms of prediction score, an initial prediction is done to compare the usage of the L_1-norm and the L_2-norm. Luckily, no major change in terms of recall, prediction and F1-score can be seen. The L_2-norm performs minimally better than the L_1-score.

Another interesting distance metric is suggested, namely the Mahalanobis distance. This distance metric would allegedly work better than any L_k-norm distance and was worth it to test it. This idea was dropped for two reasons. First of all, no literature was found regarding a study in which the Mahalanobis worked better than any L-norm distance (or at least the L_2-norm) in a high dimensional setting. The assessment of this distance metric was still worth the try. Nevertheless, the Mahalanobis distance requires the inverse of the covariant matrix of point A and B. The inverse could not always be calculated because the covariant matrix of A and B was sometimes singular whereas calculating an inverse would require a non-singular matrix.

As the Manhattan distance and Euclidean distance do not produce different results. The Euclidean distance was chosen as the distance metric for this experiment, because the Euclidean distance is the 'normal' distance metric used in mathematics.

5.3. Validating distances

To test whether assumption (2) is true, the entire data set, including the training set used to create the clusters, is used to calculate the average distance from each data point to its cluster centroid per cluster per label. See table 5.1. The average distance from the data point to its cluster centroid for `TRUE`-labeled data is greater than the average distance from the data point to its cluster centroid for `FALSE`-labeled data. Thus, in some way, assumption (2) is true because vulnerable data points are farther away from the centroid than non-vulnerable data points. If this assumption is true, it is more likely that this distance-based anomaly detection method would produce good prediction results.

Cluster	`TRUE`	`FALSE`	Combined
0	7.378	5.983	6.425
1	3.443	2.596	2.757
2	44.492	22.145	32.661
3	123.935	0	61.967
4	29.454	6.405	11.643
5	9.265	7.978	8.548
6	4.09	3.902	3.964
all	5.758	3.652	4.215

Table 5.1: The distance per label per cluster

It should be pointed out that the average distance for `FALSE`-labeled data in cluster 3 is 0. This cluster was empty after fitting the training data to the model. However, a `FALSE`-labeled data point that was not in the training set but in the entire data set is assigned to cluster 3. The distance from the centroid to this data point is 0. Also, same holds for one `TRUE`-labeled data point. The distance from this data point to the centroid was 123.935.

5.4. Model evaluation

To evaluate the prediction model, use the k-means algorithm to assign the data points of the test set to one of the clusters mentioned above. Now, If the data point of the test set is within the radius of the assigned cluster i.e. the distance of the new data point to the centroid of the assigned cluster is equal or less than the radius of that same cluster, then the new data point is an inlier and should be labeled as `FALSE`. In the case that the distance is greater, then the dissimilarity between the new data point and the data points within the assigned cluster is sufficiently high, that the new data point should be labeled as `TRUE`. Figure 5.1 illustrates this idea with an example.

For a performance metric, we have chosen both recall and precision, as we think it is important to find find all vulnerabilities and to raise as less as possible false alarms. We also look at score that combines the two, namely F1-score.

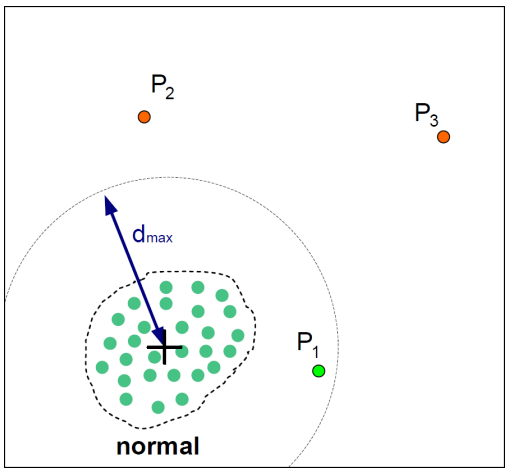


Figure 5.1: A k-means distance based outlier detection. In this example, P_1, P_2 and P_3 are three new data points. P_1 is an inlier while P_2 and P_3 are outliers [36].

5.5. Results of its application and its interpretation

In the previous section, we have described our clustering based on our training set. To evaluate this model, we use our test set to test whether vulnerable and non-vulnerable data points can be distinguished based on similarity. The training set and test set is a 80-20 split. The test set contains 752, from which 526 are non-vulnerable and 226 are vulnerable. See Appendix A for a list of formulas classification scores. See figure 5.2 and 5.3 for the results.

cluster	accuracy	recall	precision	f1-score	tn	fp	fn	tp	total data points
0	0.659	0.000	0.000	0.000	29	0	15	0	44
1	0.795	0.000	0.000	0.000	321	1	82	0	404
2	0.571	0.000	0.000	0.000	4	0	3	0	7
3	1.000	*	1.000	*	0	0	0	1	1
4	1.000	*	*	*	1	0	0	0	1
5	0.525	0.067	0.667	0.121	30	1	28	2	61
6	0.594	0.000	0.000	0.000	139	0	95	0	234
overall	0.701	0.600	0.013	0.026	524	2	223	3	752

(a) Test set results. Note that some results cannot be calculated as it requires to divide by zero.

cluster	accuracy	recall	precision	f1-score	tn	fp	fn	tp	total data points
0	0.665	0.017	1.000	0.034	112	0	57	1	170
1	0.772	0.000	0.000	0.000	1554	1	458	0	2013
2	0.553	0.105	1.000	0.190	19	0	17	2	38
3	1.000	1.000	1.000	1.000	1	0	0	1	2
4	0.826	0.200	1.000	0.333	18	0	4	1	23
5	0.551	0.032	0.833	0.062	180	1	150	5	336
6	0.638	0.000	0.000	0.000	757	0	430	0	1187
overall	0.703	0.009	0.833	0.018	2641	2	1116	10	3769

(b) Entire data set results. Note that the training set data is processed in these numbers.

Table 5.2: Anomaly detection results with 7 clusters.

cluster	max radius to centroid	training set data points amount
0	18.138	82
1	14.842	1232
2	42.109	14
3	0.000	0
4	35.743	13
5	26.614	149
6	23.213	617

Table 5.3: Clusters meta-data

The anomaly detection method produced awful results. Our first hypothesis is that the number of clusters is not optimal. After attempting the same steps with $k = 8$ and $k = 15$, the results do not improve substantially. To see what k produces good results, we train the k-means model iteratively starting from $k = 25$, increasing the number of clusters by 25. Afterwards, we proceed with the anomaly detection step. Note that maximum K cannot exceed the number of data points in the training set (2114) and therefore the maximum is chosen as $k = 2100$. This heuristic approach was not chosen before as we thought it was better to reason first what K to choose instead of deriving it from and being independent of the results. This iterative model training is run 10-fold.

From figure 5.2, we derive that the best amount of clusters is $K = 900$ in terms of F1-score. We can also see that the less clusters you have, with respect to the top at $k = 900$, the worse the performance is. Higher than $k = 900$

does not give you any better results, but slightly worse. The F1-score starts to converge at $k = 1950$, the results do not change anymore. Looking at how many are labeled positive and negative by the algorithm in figure 5.3 and 5.4, we can see that the larger the amount of clusters, the less negatives we have. The opposite is true for positives, the larger the amount of clusters, the more are labeled positive.

This observation can easily be explained. Starting from $k = 25$, each clusters will have a certain amount of data points. As the number of clusters grow, the average amount of data points in each cluster drops. This will lower the radius of each clusters. This in turn will label more data points as anomaly, as the area of the circle around the centroid (which is created by the radius) shrinks and therefore more data points from the test set will fall outside of the circle. See figure 5.5. To support this claim, the average radius and the average amount of data points in each cluster are shown in figure 5.6.

It can be seen that almost all data points are labeled negative at $k = 25$. At $k = 1950$, from this k is where all data points from the test set is labeled as positive. This is due to all clusters having one or two data points while the rest are empty. Clusters with one data point have a radius of either 0 or a number raised to the power of -16 or higher, which is practically 0. Of course, any data points assigned to this cluster will be labeled as vulnerable, as the data points are distinct and therefore the distance must be higher than the zero radius. Clusters with two data points have a small radius too, as these points are really close to each other and accordingly any other points assigned to this cluster will fall outside the circle created by the radius and will be labeled as vulnerable.

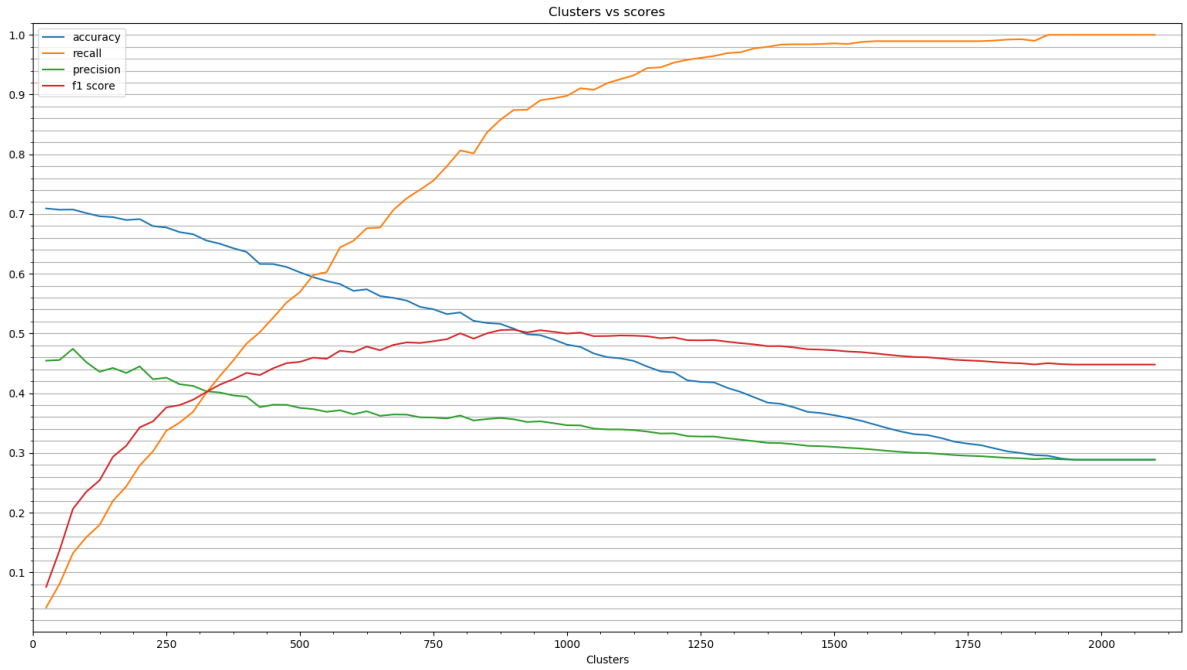


Figure 5.2: Clusters vs accuracy, recall, precision and F1-score.

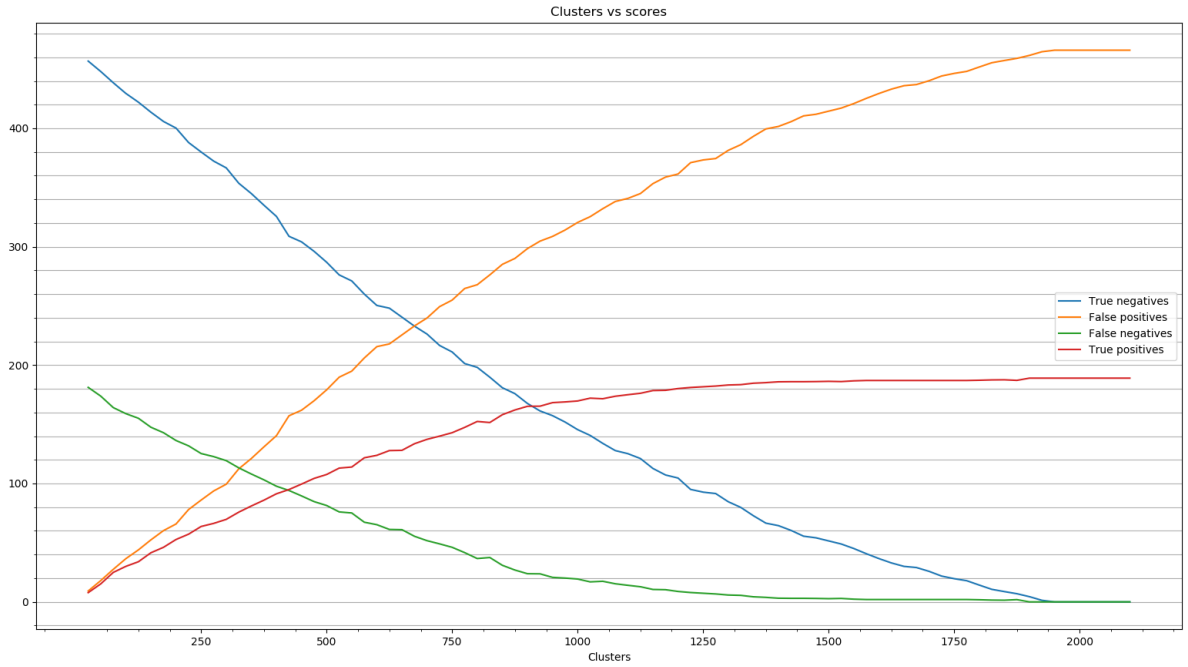


Figure 5.3: Clusters vs true negatives, false positives, false negatives and false positives.

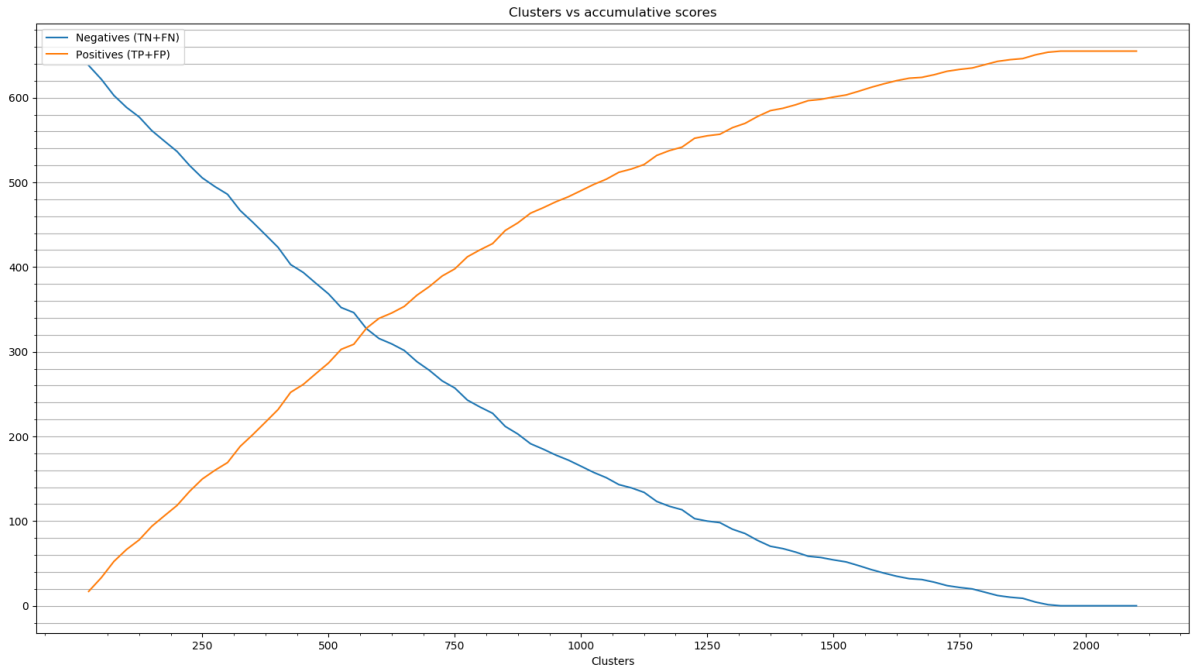


Figure 5.4: Clusters vs negatives and positives

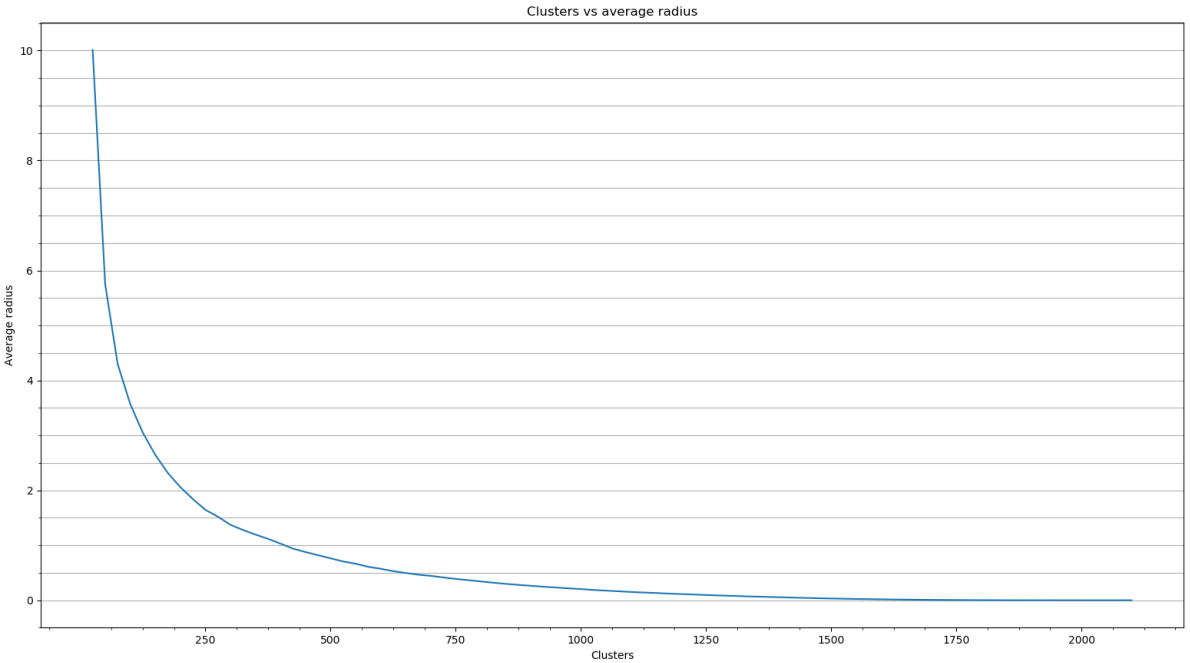


Figure 5.5: Clusters vs average radius

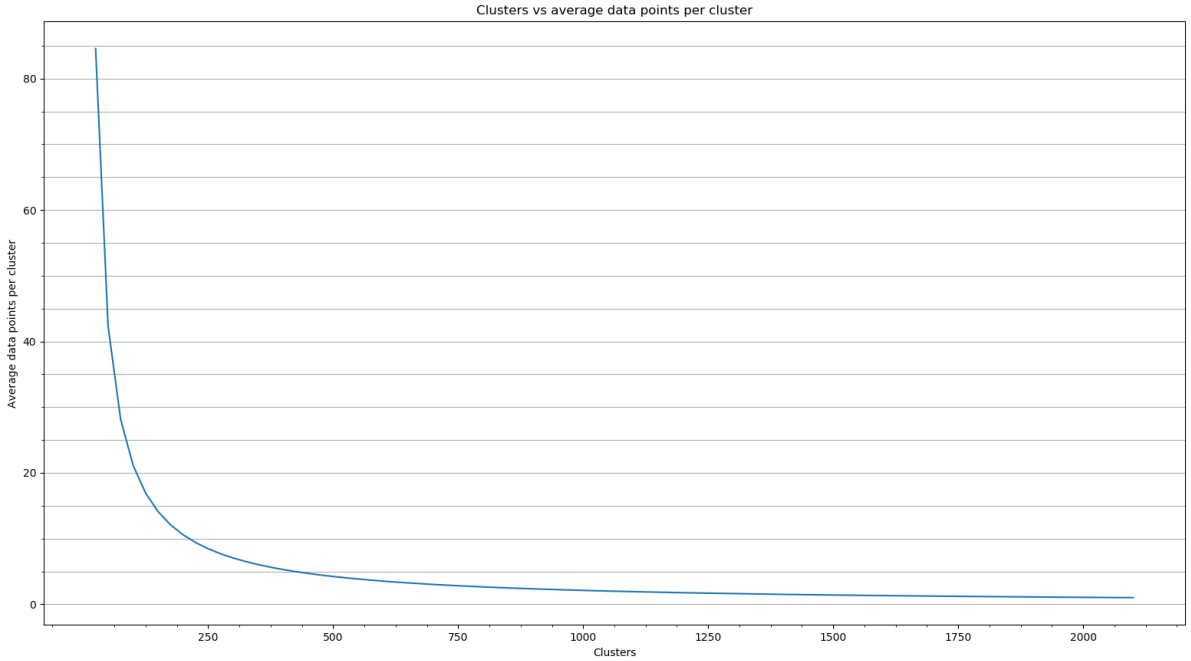


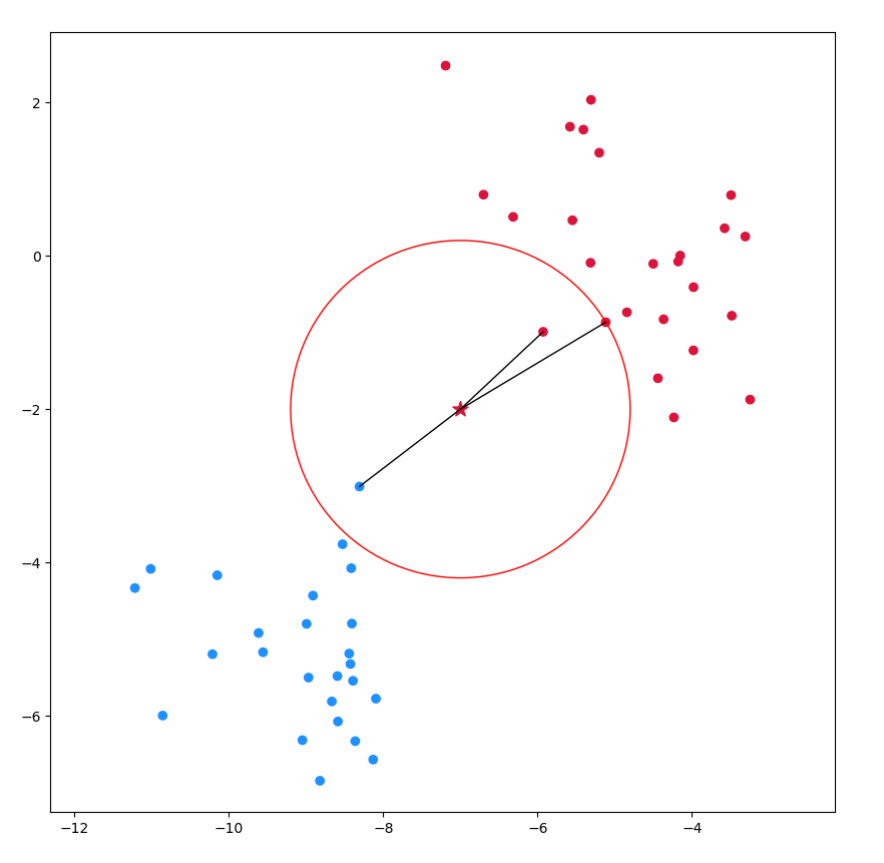
Figure 5.6: Clusters vs the average amount of data points per clusters. When non-empty clusters do not count, the average does not change considerably.

5.6. k-nearest neighbor: its application and results

As all attempts with k-means failed, our initial conclusion to the experimentation of clustering and detecting anomalies was that either these methods are deemed **not trustworthy** with respect to this data set, or the features of the data set were **not distinctive** enough to build a prediction model with. However, the idea of using another algorithm arose when looking at the figure 5.2. As can be seen, the F1-score becomes higher when the amount of clusters becomes higher. When the amount of clusters is approaching the amount of data points, every data point becomes a cluster on its own. This makes k-means look similar to the k-nearest neighbor algorithm (See Intermezzo 3). Thus, in an attempt to have better results, we use the k-nearest neighbor algorithm.

Intermezzo: k-nearest neighbor algorithm

The k-nearest neighbor (often shortened as k-NN) is a simple classification algorithm. In its simplest form, the k-nearest neighbor algorithm finds k data points that are the closest to the data point in question, where k is the parameter of the algorithm. It then classifies the data point by giving it the class of the majority of its k-nearest neighbors. In the example, the starred point is given the color red as the majority of its nearest neighbors are also red.



Intermezzo 3: k-nearest Neighbor

The distance-based anomaly detection method is optimized for k-nearest neighbor and it works as follows. For each data point in the test set, we find its nearest neighbors. If the average distance from the data point in question to its neighbors is higher than a certain threshold, then we label the point as vulnerable. There are two parameters for this method, the amount of neighbors k and the distance threshold d.

To find the best scores, we find the best combination of k and d heuristically again, from $k = 1$ to $k = 99$ inclusive and from $d = 0.1$ to $d = 2.0$ inclusive with steps of 0.1. By running it ten-fold, the F1-score has, to

our surprise, risen immensely from 0.026 ($k = 7$)/0.506($k = 900$) using k-means, to 0.820 ($k = 19$, $d = 1$). As there are $99 \times 20 = 1980$ outcomes, only the most interesting ones are shown in table 5.4. These results are quite surprising, as, at the time of writing this part nearing the end of this research, the best results of k-means and several of the explored supervised learning methods by Dinesh Bisesser[1] did not exceed an F1-score of 0.62.

k	d	accuracy	recall	precision	f1	tn	fp	fn	tp	comment
19	1	0.717	0.965	0.713	0.820	101	365	33	908	highest F1-score
53	1.6	0.727	0.916	0.739	0.818	161	305	79	862	highest accuracy
3	1.7	0.600	0.578	0.766	0.659	300	166	397	544	highest precision
1	1.8	0.523	0.414	0.765	0.538	346	120	551	390	highest precision with $k = 1$ and second highest precision
1	0.1	0.690	0.973	0.690	0.808	55	411	25	916	highest F1-score and highest accuracy with $k = 1$

Table 5.4: k-nearest neighbors results

To make sure that this new method is properly implemented, we manually verify multiple instances whether the labeling step was correct. To show this verification, for convenience by not comparing with too many neighbors, we use $k = 1$ and $d = 0.1$ (see table 5.4). In table 5.5, you can find two examples, one for a correctly labeled non-vulnerable data point and one for a correctly labeled vulnerable data point. The euclidean distance between the training data point and test data point is calculated by $d = \sqrt{\sum_{i=0}^{n} (X_i - Y_i)^2}$. The distances returned are 3.532 and 0.071 for the true negative and true positive respectively, where the former is greater than $d = 0.1$ and the latter is less or equal than $d = 0.1$. The other data points for which their distances are calculated, but are not shown is this report, were also correctly labeled. This shows that the method is correctly implemented and that the labeling is free from error.

5.7. Summary

Assuming that different groups of `FALSE`-labeled data are similar and can therefore be clustered together, and that `TRUE`-labeled data have different characteristics that `FALSE`-labeled data, we label our test data with an anomaly detection technique with the Euclidean distance. If a data point falls outside of the cluster's maximum radius, defined by the distance to the farthest point of that cluster, then the data point should be labeled as vulnerable.

The results were terrible. By changing the number of clusters in k-means does not change the results. Figure 5.2 shows a range of clusters and their score. It seems that any amount of clusters shows poor scoring. Using the maximum amount of clusters, that is the number of data points in the training set, shows an F1-score that is close to the maximum F1-Score at $k = 900$. When all data points are a cluster of its own, the k-means algorithm does not differ much from the nearest neighbor algorithm.

In a last effort to get good results, the nearest neighbor algorithm was used. Whenever the average distance from a data point to its nearest neighbors is greater than a certain threshold, we would label that data point as vulnerable. We tried different numbers of nearest neighbors and distance. When the optimal number of nearest neighbors and distance is chosen, it surprisingly shows an amazing F1-score of 0.82.

[1] Parallel research, see research scope in Chapter 1

#	train	test
0	2.654092	3.148353
1	-0.184283	-1.398478
2	1.855938	2.048891
3	-0.471544	-0.431013
4	0.625968	0.335766
5	0.172038	-1.718505
6	-0.812569	-0.382974
7	-1.199861	-0.566007
8	0.868695	-0.566791
9	-0.639004	0.106911
10	0.308658	0.74148
11	-0.200696	-1.131402
12	-0.574335	-0.22328
13	0.071989	-0.251046
14	-1.42926	-0.351019
15	0.695024	0.928444
16	0.395405	-0.084388
17	-0.431766	-0.19156
18	-0.645027	-0.306464
19	0.004216	-0.277832
20	0.522976	-0.081982
21	-0.686034	-1.27753
22	-0.746211	-0.367443
23	-0.376604	-0.461889
24	-0.983789	-0.943492
25	0.061999	-0.039764

(a) true negative: $3.532 > 0.1$

#	train	test
0	-1.202651	-1.155791
1	0.405258	0.442477
2	0.423843	0.399467
3	-0.248024	-0.240175
4	-0.172702	-0.185062
5	0.34141	0.347261
6	0.275438	0.271968
7	-0.401277	-0.402339
8	0.066992	0.070529
9	-0.387046	-0.379732
10	0.223239	0.226352
11	0.382169	0.380036
12	-0.477463	-0.480214
13	0.300985	0.306468
14	-0.187795	-0.195942
15	0.031003	0.032885
16	-0.07114	-0.057502
17	0.325793	0.325064
18	-0.202915	-0.200145
19	0.132238	0.125033
20	0.06242	0.057106
21	0.211161	0.203346
22	0.056429	0.050589
23	0.382387	0.378597
24	0.073666	0.068738
25	0.053764	0.060805

(b) true positive: $0.071 < 0.1$

Table 5.5: Example of two correctly labeled data points with k-nearest neighbor

6

End

In this chapter, the final conclusion is given together with the discussion about this research, its future work and a reflection part of the main author of this report.

6.1. Conclusion

In this section, a summary of the experiment is given, the interpretation of the results is described and the research questions are answered.

6.1.1. Summary

In the literature section, it is shown that quite some researchers are experimenting on how software vulnerabilities can be detected in an automated fashion using machine learning on software metrics. The methods of these researches are shown to be successful. We applied some of these techniques to this research, like Principal Component Analysis, k-means clustering, anomaly detection and so on.

Code was extracted from Fortify, a static code analysis tool where potential vulnerabilities have been investigated by a pentester whether it is truly a vulnerability. The code consists of meta-data, namely the vulnerability type, the exact line of code of the potential vulnerability, the risk level etc. Using an open-source tool, each of the files that contained code was used to calculate the corresponding software metrics. A data set was then created by combining these metrics and the meta-data.

The data set was cleaned by removing what was not useful to our machine learning algorithm used later on, and analyzed to get more insight of the data itself. The results of the analysis was that the problem is non-trivial, two thirds of the data are duplicates and some features are (highly) correlated. To avoid the curse of dimensionality problem as discussed in the literature section, the Principal Component Analysis was applied on our data set, resulting in a reduction from 43 features to 26 principal components, while keeping 95% of the variance of our cleaned set. These 26 principal components was used to do our clustering.

To cluster our data in distinct groups, k-means was used to divide our data into clusters. The k-parameter was chosen by using the elbow method on the variance of the created clusters. These data turned out to be good enough clustered. By choosing $k = 7$, the data was divided in a logically. For example, data with relatively extremely high values are clustered together, while data with mostly zeroes were put in a cluster. This clustering was validated by manually inspecting the data of the clusters and the source code represented by the data.

The anomaly detection step consisted the following. For each cluster, find the maximum distance from the centroid to each data point. This distance was defined as the radius of the centroid. Because the training set consisted only `FALSE`-labeled data, each data point that was within the cluster's radius is considered not vulnerable. When testing the model, the test set consisted both `TRUE`-labeled and `FALSE`-labeled data. All data point was assigned to one of the clusters created during the clustering step. They were assigned based on their similarity with the centroid of the cluster and therefore all other points in the same cluster. Still, if a data point fell outside of the

radius of its assigned cluster i.e. the distance from the data point to the cluster is greater the radius, it meant that the distance from the data point is greater than any other distance from the centroid to any other data point of the same cluster. In other words, the data point was more dissimilar than any other data point in the same cluster. In chapter 5, an assumption based on a initial intuition was made, that `TRUE`-labeled data are different (dissimilar) to `FALSE`-labeled data. Under this assumption, the data points that fell outside of the radius of its cluster should've been labeled vulnerable (`TRUE`), while data points that was inside of the radius of its cluster should've be labeled as non-vulnerable (`FALSE`). Using this outlier detection technique, however, gave us unquestionably poor results.

What falls outside the scope of answering the research questions was the usage of the k-nearest neighbors algorithm. The best results when using k-means was when the number of clusters was almost the number of data points. This behavior would be similar to k-nearest neighbor, hence we used that algorithm. The labeling worked as follows. If the average distance from the data point in question to its k-nearest neighbors was higher than a certain threshold, that data point was labeled as vulnerable, otherwise it was labeled as non-vulnerable. This method produced an F1-score of 0.82.

6.1.2. Answering the research questions

RQ1(a) What can we learn by analyzing the data set?

RQ1(b) What are the features of the data set and what subset of these features should be used for the prediction model?

- We learned from analyzing the data set that the problem is non-trivial, as there was not one column that has a high enough correlation to predict the label. There were also some columns that are (highly) correlated, so applying PCA afterwards was a good idea. We also learned that two thirds of the data set were duplicates because potential vulnerable library files, used across multiple projects, were extracted multiple times. Some rows also were also vulnerable en non-vulnerable at the same time. This was possible was Fortify would find multiple potential vulnerabilities within the same file, where the pentesters have validated at least one of them were truly vulnerable while at least one of them were not.
- The 48 features we started with were *project file class type cbo wmc dit rfc lcom totalMethods staticMethods publicMethods privateMethods protectedMethods defaultMethods abstractMethods finalMethods synchronizedMethods totalFields staticFields publicFields privateFields protectedFields defaultFields finalFields synchronizedFields nosi loc returnQty loopQty comparisonsQty tryCatchQty parenthesizedExpsQty stringLiteralsQty numbersQty assignmentsQty mathOperationsQty variablesQty maxNestedBlocks anonymousClassesQty subClassesQty lambdasQty uniqueWordsQty modifiers warningType, lineNum, nested, vulnerable*. The features were described in chapter 3 and also in appendix A.1. We have removed 5 of these features, namely `project`, `file`, `class`, `synchronizedFields` and `linenum`. The other 43 features was used for our prediction model. As PCA was applied, each of these features have a share in the principal components.

RQ2 How good are the clusters produced by k-means?

- Initially, the $k = 7$ clusters described in chapter 4 were good, as each cluster are manually validated to be logical and reasonable. This was further confirmed by producing a parallel coordinates plot where you saw a pattern that some color would stick above others, indicating that the data points were divided by how high the numbers were. Though, it was shown that the results for this instance of clustering does not produce the best results. Even so, bad results does not necessarily mean that the clusters were not good.

RQ3 To what extent are software vulnerabilities detectable using our distance-based k-means anomaly detection method?

- As shown above in the results interpretation subsection, if the number of clusters was sufficiently high

enough, the threshold of average radius would be optimal such that the F1 score was maximized. However, this F1-score was at 0.50 at maximum. The recall and precision for this F1 score was 0.87 and 0.35 respectively. It means that for every true vulnerability, there is a probability of 0.87 it would be found and for every data point labeled as vulnerable, there was a probability of 0.35 that it is correctly labeled. The recall was high enough, but the precision was too low, rendering this method overall as bad.

RQ: Can we predict software vulnerabilities, in an unsupervised learning setting, better than the Fortify by using the k-means algorithm?

- Using k-means in how it was described in our report, the answer is probably. The k-means method sure was too unreliable, either the precision was high but the recall too low or the recall was high but the precision was too low. As there was only a precision score available for the Fortify results, only the precision scores could be compared. For all numbers of k in k-means, the precision was always higher than 0.28, while the precision of Fortify was 0.02 as stated in the problem statement section in chapter 1. It means that in terms of precision, our method works better than Fortify.

  However, this does not mean that our method performs better than Fortify overall. For instance, there is no recall to compare. But do note at $k = 950$, when the F1-score is the highest, our recall was 0.88. That is an astounding recall and not easily to be exceeded.

6.2. Discussion and future work

The conclusion of the main research question was that our method using k-means works better than Fortify in one aspect. However, judging our method overall, the k-means method was not that good either. This conclusion came to be due to the observation that the radii of each cluster decreases as the amount of clusters were larger, so that means data points are more likely to be labeled a vulnerable. This does not seem to good logic behind it, as clusters should have generalized characteristics that you group the data points into. The highest F1-score we got was when we initialized k-means with 950 clusters. That is too many and we doubt that it is logical to group clusters into at least 950 characteristics. Even so, the highest F1-score was still to low. This gave us a reason to think the features were faulty.

This reasoning was further 'confirmed' by the parallel research done by Dinesh Bisesser, who is doing the same research with multiple supervised machine learning algorithms, shows that his results were also unsatisfactory (maximum 0.61 F1-score). We took his research results into account and concluded that the features of the data set are not distinctive enough to do any machine learning predictions, as multiple supervised or unsupervised algorithm fail doing so.

But after using the nearest neighbor method, it showed that using the right amount of nearest neighbor and distance threshold, the F1-score was a 0.82, higher than any F1-score that we or Dinesh Bisesser could score. If the F1-score could be this high, then it probably is not that the features are too faulty, rather it is the method/techniques that were applied that was faulty. The k-means algorithm and the distance-based anomaly detection technique used were chosen due to its popularity and to its simplicity, but its simplicity might be the reason that the results were unsatisfactory.

There is quite some part of this research that needs further exploration.

First, the K-means algorithm is, as stated in the literature section, the most popular yet one of the simplest clustering algorithm. It was chosen due to its simplicity. The k-means algorithms produced to the human eye seemingly good results, as mentioned in chapter 4 and here above again. However, it does not mean that the algorithm clustered the data in the best way possible. To test this, multiple clustering methods are required to be used and compared. Such clustering methods are DBSCAN, Hierarchical clustering and so on. Testing different clustering methods can show which clustering methods work the best with the data set provided in this research. Because using clustering algorithms other than k-means falls outside of the scope of this research, testing different clustering methods is left for future work.

Another technique to evaluate is the anomaly detection technique. For each centroid of the cluster, define the radius of a cluster as the the max distance from the centroid to the data points in the cluster. Every new data point appointed to this cluster, but falls outside this radius will be marked as an anomaly. This technique is also

chosen for its simplicity, but unfortunately, this technique is maybe also too simple. One of the reason is that the direction from centroid to data point was not taken into account. In other words, how the data points differ from each other is irrelevant. The distance between two data points with 100 and 200 lines of code and both 1 return statement, might be the same distance as two points both with 100 lines of code where one point has 1 return statement and the other one has 2 return statements, if we assume that all other numbers are the same. In the end, the euclidean distance is taken so the outcome of distance is the same for both the examples. The scaling is dependent on other points in the same column. But when assuming the scaling is done such that these two examples have the same distance, this will create a problem as lines of code might be a more useful feature than the number of return statements in terms of labeling the data. This is further shown by the conclusion stated above that there there exist a correlation between lines of code and the probability that code is vulnerable.

There was a long discussion about how to handle duplicates. Duplicates existed because for the same file, multiple potential vulnerability were found by Fortify. These potential vulnerabilities were sometimes vulnerable and non-vulnerable for the same file. This produced duplicates in the data set with a different label. Due to it being contradicting, it was a reason to change all these non-vulnerable labels to a `TRUE`-labeled data point if there would exist a vulnerable data point for the same numbers. Then, it was also possible that duplicates existed for files in different projects. At first, these duplicates were kept in the data set because we thought it was merely a coincidence that different projects have files with the same (generic) name and same row. During the phase of testing our model, the duplicate problem was discussed again. By removing the duplicates, the recall and precision would significantly drop. This was not desired so the duplicates were kept. After testing all our models and getting more and more inconsistent results, the code represented by these duplicate data points were further inspected to determine whether a mistake have been made by keeping them in the model. It turned out to be that different projects uses the exact same code (as code from the same library), therefore it had the exact same file name and numbers in the data set. Duplicates with 157(!) same instances were found in our data set. As mentioned before, the results were inconsistent due to the fact that the more clusters the data points were divided into, the better the results. These best results in terms of recall and precision were produced with $k = 1500$, It was illogical to have this many clusters by not being able to explain what each cluster mean as we did in the clustering section of this report. Also, by having too many clusters in contrast to the amount of data points in the data set, the model would not capture the patterns of our data set. This would contradict the essence of machine learning to capture patterns in a data set, hence it would not be useful to have such a model. It had us thinking that these results that improves with a larger k was because the centroids compared with the same data points over and over (we kept in mind that we had these duplicates in the data set). These duplicates were ultimately removed. For a long time no progress had been made due to this problem. This could all have been avoid at the beginning if more attention had been paid to this matter.

Days before submitting this report, when generating certain plots again to put into the report, the t-SNE plots were generated again. To our surprise, this time the t-SNE plots showed something else than what we had before. The data points were grouped together this time. See figure 6.1. Though there is no time to find out what went wrong now, what can be concluded is that the clusters can be grouped into 8 clusters, this can be seen from $p = 70$ on. After quickly clustering using k-means with $k = 8$, you can see that the clustering inconsistent with the t-SNE graphs as, some lines have multiple colors. See figure 6.2. After executing k-means for a hundred times, the clustering stays inconsistent. This makes us think that k-means is not clustering these data points optimally and hence it could be the reason why the k-means method failed. Maybe with a little bit more data pre-processing will get k-means to work better. Otherwise, clustering the data according to the t-SNE graphs, can be done manually or choosing an algorithm that is consistent with these t-SNE graphs.

As for the results, the k-means scores is unsatisfactory, but the k-nearest neighbors score on the other hand is much better. It has an excellent recall of 0.965 and a good precision of 0.713. The precision could be better though. It means that 28.7% of all positives found is a false alarm, which is still a lot. Another thing to mention is that the amount of neighbor and distance threshold chosen for the optimal results ($k = 19$, $d = 1$) might only work on these particular set of data points. It might not work if the data set is extended with new data. There has yet to be a reason given why exactly these number of neighbors and distance threshold give such good results and also why this method works in general, but that also can be done in the future.

Aside from that, there are also other limitations to this research. The data set could be inconsistent. As we extracted the code from Fortify and checked whether potential vulnerabilities were set to *approved* or *not approved*, we saw comments attached to it that were posted by employees of ING. We saw their corporate key (personal identifier) as identification, but we did not see any names, nor did we see whether they were developers or pen-

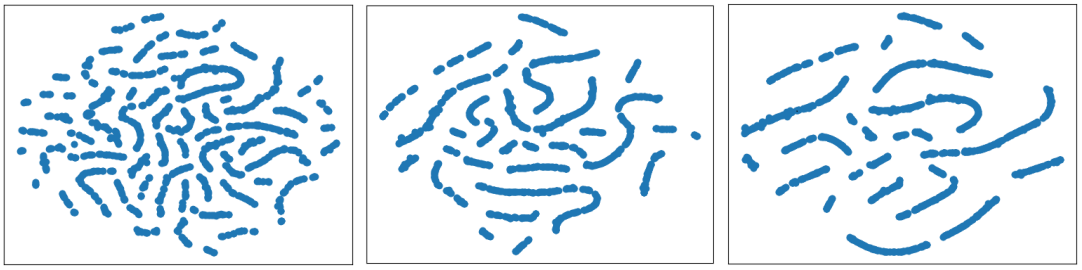
testers. It could be the case that a developer gave it some tag, but it was incorrect and it had yet to be approved by a pentester.

Also, we only have a small portion extracted of all potential vulnerabilities shown in Fortify. There were 930 projects at that time, but most of the potential vulnerabilities within those 930 projects did not have *approved* or a *not approved* tags. It could be the case that we did not capture enough the potential vulnerabilities to have the pattern in our data set that we would like to our model find. The captured file were not random, in a sense that they are scattered across all projects. Projects that were chosen to be extracted, had all their code extracted, where most of the other projects have nothing extracted from. It would be better if we had files from all projects. Then our data set would have more chance to have this pattern we are trying to find. But this was not possible as only a few number of projects had someone validate if it was vulnerable.

6.3. Reflection

For the entire report, I, the main author of this report, have used 'we' to describe our work as myself and everybody who has helped me. For this section, I will explicitly talk about only myself, as part of the a reflection on myself.

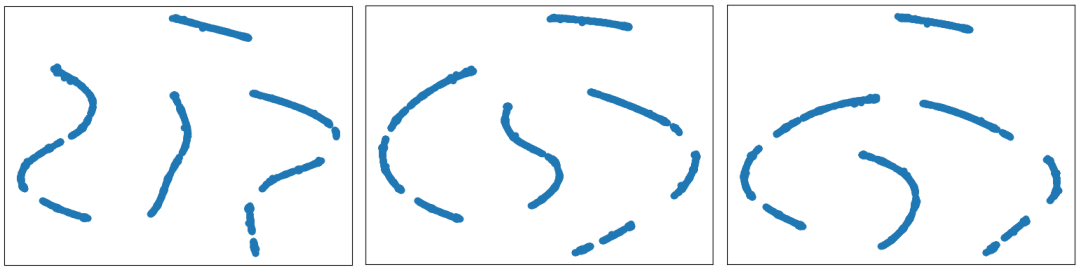
As this experience is quite new for myself, many mistakes were made by myself throughout the research. Though my code works, it is written very bad and monolithic. When I wanted to execute code from months ago with a slight modification, It would take very long sometimes to change code. It took me a very long time to realize that the duplicate in the data set come from exactly the same code. At the beginning, it was chosen to keep the duplicates, but that was before realizing the code was exactly the same, as already mentioned in the discussion section above. Also, documenting what I did would be a good idea, as I could not find files or what I have done in the past. If the t-SNE graphs were good at the beginning, the process of the report would probably be different as the k-means does not cluster the data point in line with the t-SNE graphs. Though this might not change the end results, but as said before, this can be found out in future work. I also needed a lot of help from my supervisors in my opinion. Conducting this research would be a lot more difficult without their help, describing to me how I should outline my research and this report to begin with, how to prove my thinking and so on. As this is my first huge report, the writing could probably be better so the reader can read my report more comfortable. What I would have done differently is also not to rely too much on others and take more time to study so you can make the decisions yourself. Look more at the work of other people (in this situation it would be thesis reports of other students) so you have a general idea of what you should do yourself. Though I have done it, I did not do it enough.



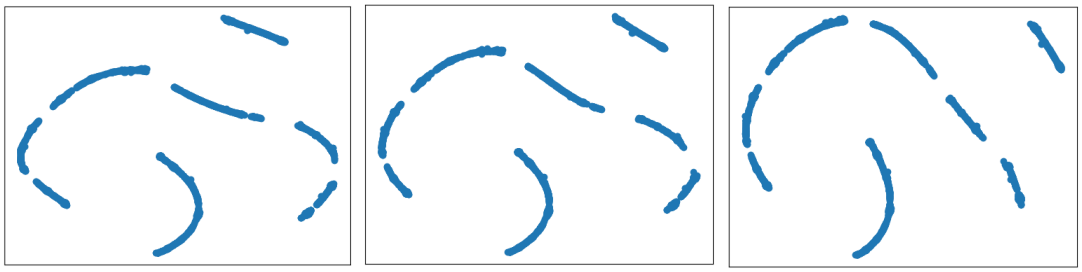
(a) $p = 5$ (b) $p = 10$ (c) $p = 15$

(d) $p = 20$ (e) $p = 25$ (f) $p = 30$

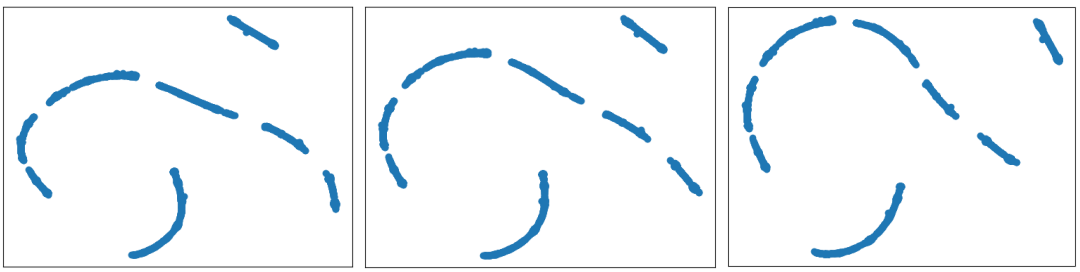
(g) $p = 35$ (h) $p = 40$ (i) $p = 45$



(j) $p = 50$ (k) $p = 55$ (l) $p = 60$

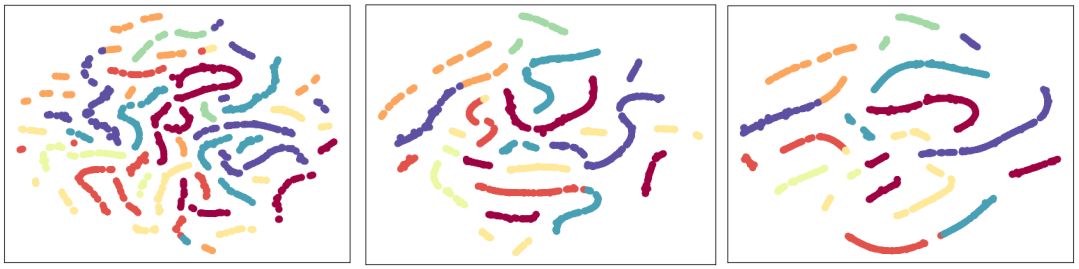


(m) $p = 65$ (n) $p = 70$ (o) $p = 75$

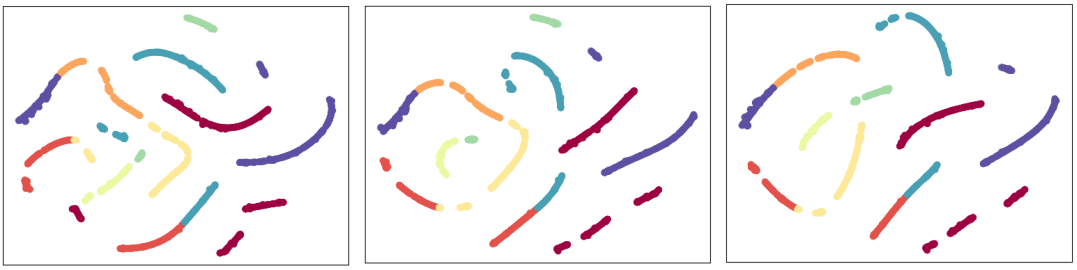


(p) $p = 80$ (q) $p = 90$ (r) $p = 90$

Figure 6.1: Another t-SNE visualizations

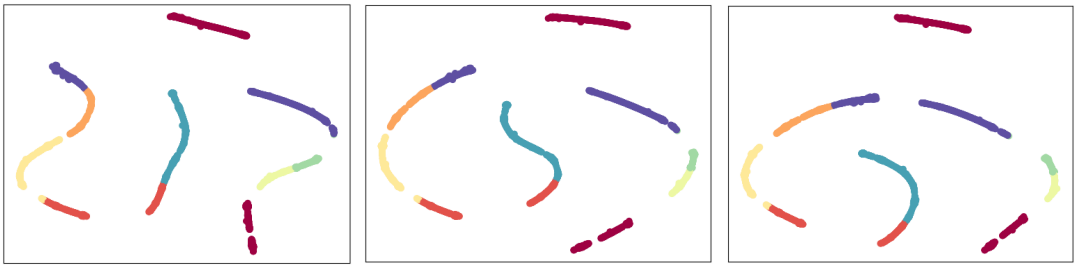


(a) $p = 5$ (b) $p = 10$ (c) $p = 15$

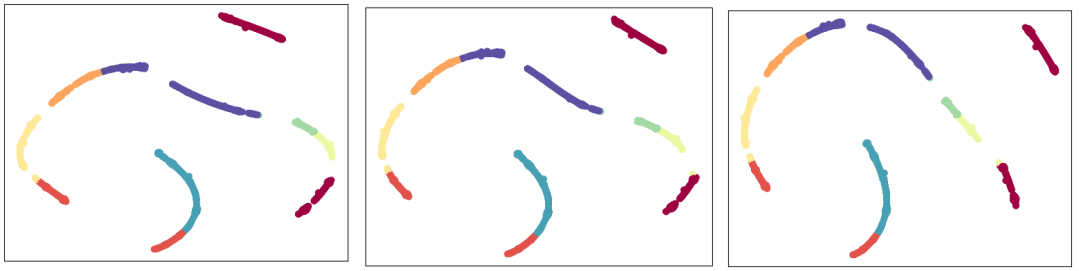


(d) $p = 20$ (e) $p = 25$ (f) $p = 30$

(g) $p = 35$ (h) $p = 40$ (i) $p = 45$



(j) $p = 50$ (k) $p = 55$ (l) $p = 60$



(m) $p = 65$ (n) $p = 70$ (o) $p = 75$

(p) $p = 80$ (q) $p = 90$ (r) $p = 90$

Figure 6.2: t-SNE with coloring based on clustering using k-means with $k = 8$

A

Appendix

Appendix A

Accuracy (percentage of correct labeling):

$$\frac{TP+TN}{TN+FP+FN+TP}$$

Recall (percentage of all positive examples found):

$$\frac{TP}{TP+FN}$$

Precision (percentage of correctly labeled positives):

$$\frac{TP}{TP+FP}$$

F1-score (weighted average of recall and precision):

$$F1 = 2 \cdot \frac{(\text{precision} \cdot \text{recall})}{(\text{precision} + \text{recall})}$$

Figure A.0: Classification scores

Appendix B

Metric	Description
CBO	Coupling between objects: Counts the number of dependencies a class has.
DIT	Depth Inheritance Tree: it counts the number of parent a class has.
Number of fields	Counts the number of fields.
Number of methods	Counts the number of methods. Specific numbers for total number of methods, static, public, abstract, private, protected, default, final, and synchronized methods
NOSI	Number of static invocations: Counts the number of invocations to static methods.
RFC	Response for a Class: Counts the number of unique method invocations in a class.
WMC or McCabe's complexity	Weight Method Class: It counts the number of branch instructions in a class.
LOC	Lines of code: it counts the lines of code, ignoring empty lines.
LCOM	Lack of Cohesion of Methods.
Quantity of returns	The number of return instructions
Quantity of loops	The number of loops (i.e., for, while, do while, enhanced for)
Quantity of comparisons	The number of comparisons (i.e., == and !=).
Quantity of trycatches	The number of trycatches
Quantity of parenthesized expressions	The number of expressions inside parenthesis
String literals	The number of string literals (e.g., "John Doe"). Repeated strings count as many times as they appear
Quantity of Number	The number of numbers (i.e., int, long, double, float) literals
Quantity of Math Operations	The number of math operations (times, divide, remainder, plus, minus, left shit, right shift)
Quantity of Variables	Number of declared variables k
Max nested blocks	The highest number of blocks nested together
Quantity of Anonymous classes, subclasses, and lambda expressions	Number of anonymous classes, subclasses and lambda expression.
Number of unique words	Number of unique words in the source code.
Modifiers	public/abstract/private/protected/native modifiers of classes/methods.
Usage of each variable	How much each variable was used inside each method.
Usage of each field	How much each field was used inside each method.

Table A.1: Description of each column in the data set

Appendix C

Warning type	Occurences
System Information Leak: Internal	4627
J2EE Bad Practices: Threads	1615
Log Forging	627
Password Management: Hardcoded Password	480
Path Manipulation	335
Race Condition: Singleton Member Field	310
Unreleased Resource: Streams	265
Dynamic Code Evaluation: Unsafe Deserialization	192
System Information Leak	185
Missing XML Validation	185
Mass Assignment: Insecure Binder Configuration	178
XML External Entity Injection	164
Code Correctness: Byte Array to String Conversion	131
Access Specifier Manipulation	125
Insecure Randomness	125
Missing Check against Null	107
J2EE Bad Practices: Leftover Debug Code	105
Access Control: Database	90
Code Correctness: Non-Static Inner Class Implements Serializable	88
Cookie Security: Cookie not Sent Over SSL	87
J2EE Bad Practices: Sockets	86
Header Manipulation *	81
Portability Flaw: Locale Dependent Comparison	80
XML Entity Expansion Injection	66
Unsafe Reflection	65
System Information Leak: External	63
Dynamic Code Evaluation: JNDI Reference Injection	63
Code Correctness: Double-Checked Locking	62
Weak Cryptographic Hash	61
Insecure SSL: Overly Broad Certificate Trust	54
Denial of Service	52
Resource Injection	51
JSON Injection	50
SQL Injection	49
Key Management: Hardcoded Encryption Key	49
Potential for Unsafe Deserialization	47
Often Misused: Authentication	45
Null pointer dereference	43
Assignment to parameter	39
Object Model Violation: Just one of equals() and hashCode() Defined	38
Dubious method used	37
Unchecked Return Value	36
Insecure Randomness: Hardcoded Seed	35
Privilege Management: Overly Broad Access Specifier	34
Race Condition: Format Flaw	32
Unused field	31
Cookie Security: HTTPOnly not Set	30
Questionable Boxing of primitive value	29
Cross-Site Scripting: Reflected	27
SQL Wildcards	27
Unread field	25
Insecure Transport	20

Incorrect definition of Serializable class	20
Setting Manipulation	18
Inner class could be made static	18
Unreleased Resource: Database	18
Problems with implementation of equals()	17
Cross-Site Scripting: Persistent *	15
Redundant comparison to null	14
Mutable static field	14
Code Correctness: Erroneous String Compare	13
RuntimeException capture	13
Password Management: Null Password	13
SQL Injection: Persistence	12
System Information Leak: Incomplete Servlet Error Handling	12
Insecure Transport: Mail Transmission	10
Code Correctness: Non-Synchronized Method Overrides Synchronized Method	10
Weak SecurityManager Check: Overridable Method	9
Access Control: LDAP *	9
HTTP Parameter Pollution	9
JavaScript Hijacking	9
Code Correctness: Hidden Method	9
Password Management	8
Misuse of static fields	8
Server-Side Request Forgery	8
Weak Encryption: Insecure Mode of Operation *	8
Cross-Site Scripting: Content Sniffing	8
Privacy Violation	8
File Disclosure: Struts	7
J2EE Bad Practices: getConnection()	7
Header Manipulation: Cookies	7
J2EE Bad Practices: JVM Termination	7
Unreleased Resource: Sockets	7
Access Control: SecurityManager Bypass *	7
SQL Injection: Hibernate	6
Naming of command class	6
Switch case falls through	6
Password Management: Weak Cryptography	6
Suspicious use of non-short-circuit boolean operator *	6
Command Injection	5
Useless control flow	5
Dead local store	5
Inefficient Map Iterator	5
Race Condition: Class Initialization Cycle *	4
Key Management: Empty Encryption Key	4
Password Management: Empty Password	4
Encoded Data Validation	4
Open Redirect *	4
Session Puzzling: Spring	4
Bad use of return value from method	4
HTML5: Overly Permissive CORS Policy	4
Cross-Site Request Forgery *	4
Missing Check for Null Parameter *	4
Types: Using class Double	4

LDAP Injection	4
Code Correctness: Erroneous Class Compare	4
String concatenation in loop using + operator *	3
Code Correctness: Multiple Stream Commits *	3
Often Misused: File Upload *	3
Often Misused: Boolean.getBoolean()	3
Weak Encryption: Insecure Initialization Vector	3
Static: Static members in Servlets must be final *	3
EJB Bad Practices: Use of Sockets	3
Weak Encryption: Inadequate RSA Padding *	3
Weak Encryption: Byte Array to String Conversion	3
Weak Cryptographic Hash: User-Controlled Algorithm	2
Denial of Service: Regular Expression	2
Inconsistent synchronization	2
Key Management: Null Encryption Key	2
Casting from integer values	2
Questionable use of reference equality rather than calling equals	2
Unsynchronized Lazy Initialization	2
Poor Logging Practice: Logger Not Declared Static Final *	2
Useless code	2
Denial of Service: Format String	2
Uncallable method of anonymous class	2
Unsatisfied obligation to clean up stream or resource *	2
Weak Encryption: Missing Required Step *	2
Code Correctness: Call to Thread.stop()	2
File Disclosure: J2EE	2
Double check pattern *	2
Regular expressions	1
Format string problem	1
Duplicate Branches *	1
HTTP Response splitting vulnerability *	1
Do not use a CallableStatement *	1
Cross-Site Scripting: Poor Validation	1
Information Disclosure	1
Synchronization on java.util.concurrent objects	1
Dubious method invocation	1
Unwritten field	1
Bad casts of object references	1
Unsafe Mobile Code: Unsafe Public Field *	1
Unsafe Mobile Code: Access Violation	1
Server Misconfiguration: HTTP Basic Authentication	1
Header Manipulation: SMTP	1
Uninitialized read of field in constructor	1
Serializable class with no Version ID *	1
Code Correctness: Class Does Not Implement Cloneable *	1
Code Correctness: null Argument to equals()	1
Object Model Violation: Erroneous clone() Method	1
Formula Injection *	1
Useless/non-informative string generated	1
Poor Error Handling: Program Catches NullPointerException *	1
Dead Code: Empty Try Block *	1
LDAP Entry Poisoning	1
Cookie Security: Persistent Cookie	1

Table A.2: Warning types and their occurences in the data set.
* Rows with these warning types all have the `vulnerable` column as `TRUE`

Bibliography

[1]

[2] sklearn.preprocessing.standardscaler¶. URL https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler.

[3] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*, pages 420–434. Springer, 2001.

[4] Samuel Ajila and Razvan Dumitrescu. Experimental use of code delta, code churn, and rate of change to understand software product line evolution. *Journal of Systems and Software*, 80:74–91, 01 2007. doi: 10.1016/j.jss.2006.05.034.

[5] H. Alves, B. Fonseca, and N. Antunes. Software metrics and security vulnerabilities: Dataset and exploratory study. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 37–44, Sep. 2016. doi: 10.1109/EDCC.2016.34.

[6] Paul Anderson. The use and limitations of static-analysis tools to improve software quality. *CrossTalk: The Journal of Defense Software Engineering*, 21(6):18–21, 2008.

[7] Maurício Aniche. *Java code metrics calculator (CK)*, 2015. Available in https://github.com/mauricioaniche/ck/.

[8] Alexandru G Bardas et al. Static code analysis. *Journal of Information Systems & Operations Management*, 4(2):99–107, 2010.

[9] Cagatay Catal. Software fault prediction: A literature review and current trends. *Expert Systems with Applications*, 38(4):4626 – 4636, 2011. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2010.10.024. URL http://www.sciencedirect.com/science/article/pii/S0957417410011681.

[10] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346 – 7354, 2009. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2008.10.027. URL http://www.sciencedirect.com/science/article/pii/S0957417408007215.

[11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994. ISSN 0098-5589. doi: 10.1109/32.295895.

[12] Istehad Chowdhury and Mohammad Zulkernine. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In *SAC*, 2010.

[13] B.Terry Compton and Carol Withrow. Prediction and control of ada software defects. *Journal of Systems and Software*, 12(3):199 – 207, 1990. ISSN 0164-1212. doi: https://doi.org/10.1016/0164-1212(90)90040-S. URL http://www.sciencedirect.com/science/article/pii/016412129090040S. Oregon Workshop on Software Metrics.

[14] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[15] Gabriela Czibula, Zsuzsanna Marian, and Istvan Gergely Czibula. Software defect prediction using relational association rule mining. *Information Sciences*, 264:260 – 278, 2014. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2013.12.031. URL http://www.sciencedirect.com/science/article/pii/S0020025513008876. Serious Games.

[16] AA Shahrjooi Haghighi, M Abbasi Dezfuli, and SM Fakhrahmad. Applying mining schemes to software fault prediction: a proposed approach aimed at test cost reduction. In *Proceedings of the World Congress on Engineering*, volume 1, pages 4–6, 2012.

[17] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977. ISBN 0444002057.

[18] D. Hammerstrom. Working with neural networks. *IEEE Spectrum*, 30(7):46–53, 1993.

[19] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[20] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[21] I.T Jolliffe. *Principal Component Analysis - Second Edition*. Springer-Verlag, 2002.

[22] Kaspersky Lab. Carbanak apt: The great bank robbery. *Securelist*, 2015.

[23] Paige Leskin. The 21 scariest data breaches of 2018, Dec 2018. URL `https://www.businessinsider.nl/data-hacks-breaches-biggest-of-2018-2018-12/`.

[24] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. Software vulnerability discovery techniques: A survey. In *2012 Fourth International Conference on Multimedia Information Networking and Security*, pages 152–156. IEEE, 2012.

[25] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[26] Ruchika Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504 – 518, 2015. ISSN 1568-4946. doi: https://doi.org/10.1016/j.asoc.2014.11.023. URL `http://www.sciencedirect.com/science/article/pii/S1568494614005857`.

[27] Steve Mansfield-Devine. The ashley madison affair. *Network Security*, 2015(9):8–16, 2015.

[28] Trust Tshepo Mapoka, Keneilwe Zuva, and Tranos Zuva. Hack the bank and best practices for secure bank.

[29] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233837.

[30] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *2008 IEEE International Conference on Software Maintenance*, pages 346–355, Sep. 2008. doi: 10.1109/ICSM.2008.4658083.

[31] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'". *IEEE Transactions on Software Engineering*, 33(9):637–640, Sep. 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.70721.

[32] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, Jan 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.256941.

[33] MicroFocus. About the analyzers. URL `https://www.microfocus.com/documentation/fortify-static-code-analyzer-and-tools/2010/SCA_Help_20.1.2/index.htm#Introduction/Analyzers.htm`. Accessed on 3rd August 2020.

[34] Microfocus. Fortify static code analyzer (sca) static application security testing datas sheet. URL `https://www.microfocus.com/media/data-sheet/fortify_static_code_analyzer_static_application_security_testing_ds.pdf`. Accessed on 3rd August 2020.

[35] Savita Mohurle and Manisha Patil. A brief study of wannacry threat: Ransomware attack 2017. *International Journal of Advanced Research in Computer Science*, 8(5), 2017.

[36] Gerhard Münz, Sa Li, and Georg Carle. Traffic anomaly detection using k-means clustering. In *GI/ITG Workshop MMBnet*, pages 13–14, 2007.

[37] J. Murillo-Morera, Christian Quesada-López, and Marcelo Jenkins. Software fault prediction: A systematic mapping study. pages 446–459, 04 2015.

[38] National Institute of Standards & Technology. The economic impacts of inadequate infrastructure for software testing, may 2002. Accessed: 27th July 2019.
Available at: `https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf`.

[39] OWASP. Null dereference. URL `https://owasp.org/www-community/vulnerabilities/Null_Dereference`. Accessed on 3rd August 2020.

[40] OWASP. Owasp top ten, 2017. URL `https://github.com/OWASP/Top10/raw/master/2017/OWASPTop10-2017(en).pdf`. Accessed on 3rd August 2020.

[41] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[42] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-238-0.

[43] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8):1397–1418, 2013.

[44] Paul Sandle. Hp enterprise strikes $8.8 billion deal with micro focus for software assets, Sep 2016. URL `https://www.reuters.com/article/us-hpenterprise-software-microfocus-idUSKCN11D2EU`. Accessed on 5th October 2019.

[45] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37 (6):772–787, Nov 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.81.

[46] Lindsay I Smith. A tutorial on principal components analysis. Technical report, 2002.

[47] Laurens Van Der Maaten. Accelerating t-sne using tree-based algorithms. *The Journal of Machine Learning Research*, 15(1):3221–3245, 2014.

[48] Michael E Wall, Andreas Rechtsteiner, and Luis M Rocha. Singular value decomposition and principal component analysis. In *A practical approach to microarray data analysis*, pages 91–109. Springer, 2003.

[49] H. Zhang and X. Zhang. Comments on "data mining static code attributes to learn defect predictors". *IEEE Transactions on Software Engineering*, 33(9):635–637, Sep. 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.70706.