Improving the Usefulness of Alerts Generated by Automated Static Analysis Tools

Jakob Bleier

Radboud University Nijmegen
April 3, 2017

Supervised by:
dr. ir. Erik Poll
dr. ir. Haiyun Xu

Second Reader:
prof. dr. ir. Joost Visser
Abstract

Static analysis tools are programs that analyze software without executing it. They can be simple style checkers or follow intricate rules to efficiently find problems often overlooked by developers. Unfortunately, the alerts generated by those tools are not always correct. The high number of false positives has been found to be one of the major reasons why such tools have not been widely adopted. One approach to improve these tools is to post-process the generated alerts and only report actionable ones, i.e. true positives which will be acted upon by the developers.

In this work, we evaluate several machine-learning classifiers that use historic alert data to classify new alerts as actionable or not. We build a framework called Autobugs to collect the necessary information. It runs a static analysis tool on past revisions of a software project, saves the generated alerts and computes the lifetime-based actionability label.

This is then used to train a linear support vector machine (SVM), a non-linear SVM and a decision tree on three similar open-source forum-software projects written in PHP. We evaluate each classifiers for each project individually as well as the application of a trained model on a different project.

Based on the results, we constructed an additional classifier, which only takes into account the lifetime of an alert, classifying younger ones as actionable. It outperforms the other algorithms for our sample software-projects.
Acknowledgement

This work would not have been possible without the support of many people. It is my pleasure to thank my supervisors Erik Poll for his always helpful feedback and Joost Visser for his inspiring questions. My special thanks go to Barbara Vieira and Haiyun Xu for providing all the help and encouragement I could wish for.

I am grateful for having been part of the Software Improvement Group for a semester, where I wrote this thesis as part of my internship. It was brilliant to see such an inspiring company from the inside.

Further thanks go to my parents, who supported me with all their love, Niklas, who never refused to pick me up when I was feeling down, and Boki, the strongest fairy of them all - thank you for being part of my life.

There are no spelling mistakes in this work, merely reminders to the reader that further work is needed on improving the usefulness of static text analysis.
# Contents

Abstract .......................................................... i
Acknowledgement .................................................. ii

1 Introduction .................................................. 1
  1.1 Research Question .......................................... 2
  1.2 Thesis Overview ........................................... 3

2 Background .................................................. 5
  2.1 Static Analysis ............................................. 5
      2.1.1 FindBugs ............................................. 6
      2.1.2 Static Analysis Tool A ............................. 7
  2.2 Problems with Static Analysis .............................. 7
  2.3 Post-Processing Methods .................................. 8
      2.3.1 Definitions .......................................... 8
      2.3.2 Previous Work on Post-Processing ................ 10
      2.3.3 Limits of Previous Work ........................... 12

3 Framework for Collecting Historic Alert Data ............. 14
  3.1 Description of Framework Architecture .................. 15
      3.1.1 Workflow .......................................... 15
      3.1.2 Selecting a Subset of Revisions ................... 16
      3.1.3 Computing Lifetime & Actionable Alert Classification 17
      3.1.4 Configuration Files of Static Analysis Tools ........ 18
  3.2 Experiment 1A - FindBugs with JDOM .................... 18
      3.2.1 JDOM .............................................. 18
      3.2.2 Comparison to the Work of Heckman et al. .......... 18
      3.2.3 Analysis and Visualization of the Data ........... 19
      3.2.4 Discussion ........................................ 21
  3.3 Experiment 1B - FindBugs with Elasticsearch ............ 23
      3.3.1 Elasticsearch .................................... 23
      3.3.2 Analysis and Visualization ........................ 24
  3.4 Experiment 2A and 2B - Static Analysis Tool A with JDOM and Elasticsearch .............................. 27
CONTENTS

3.4.1 Static Analysis Tool A ......................................... 27
3.4.2 Lifetimes ......................................................... 30
3.5 Discussion of the Experiments ................................. 34
  3.5.1 Actionable Alert Metric .................................... 34
  3.5.2 Alert Lifetimes ............................................... 35

4 Feature Extraction and Classifiers ............................... 36
  4.1 Process of Building a Classification Model .................. 36
  4.2 Features ......................................................... 37
  4.3 Classifiers ..................................................... 39
  4.4 Performance Metrics .......................................... 40
  4.4.1 Evaluating the Models .................................... 40

5 Evaluation of Features and Classifiers .......................... 42
  5.1 Experiment Setup ............................................... 42
  5.2 Results ......................................................... 44
    5.2.1 Features ................................................... 44
    5.2.2 Experiment 3A: Train- and Test-Data from Same Project 44
    5.2.3 Experiment 3B: Cross-Project Performance .............. 47
    5.2.4 Experiment 3C: Custom Classifier Based on Lifetime ... 52
  5.3 Discussion ..................................................... 52

6 Future Work ................................................................ 55
  6.1 Alternative Approaches ........................................ 55
  6.2 Using Historic Alert Data ...................................... 56

7 Conclusions ................................................................ 57
  7.1 Discussion ......................................................... 58

Appendix .................................................................. 62
Chapter 1

Introduction

Writing software is similar to authoring a text. After choosing a language one has to spell the words correctly and follow rules to express a certain meaning. Sometimes the product is a reminder for oneself that nobody else will read. Other times it is an intricate contract, which needs to be very precise in order to fulfill its purpose of explicitly expressing an agreement. And sometimes poets bend and break the rules of grammar, spelling, and punctuation to create beautiful poems that others spend years interpreting.

Software usually does not fall into the latter category. While natural language has a lot of room for interpretation, programming languages have to follow a strict grammar and syntax. The main recipient of a program is the computer and it is not very good at interpreting rule-bending expressions. In fact, all it will do is exactly what it has been told. This leads to a lot of frustration and unexpected results when someone’s expectation of what a piece of code says is different from what it actually tells the computer to do.

Mistakes such as typos quickly lead to warnings, errors and the computer rejecting the program. However, when a typo results in another valid expression, then it becomes more difficult for the computer to recognize it as a mistake. For example, the sentence “Can we meet later, my grain is running late?” is grammatically correct, but makes little sense. Humans can easily infer from the context that a train was meant, but for a computer this context is difficult to access. This is why programs are often given by the author to others to check for mistakes, just like manuscripts are given to copy-editors. In software engineering, this process is called code review [2].

The reviewer, just like a copy-editor, looks for mistakes: from simple typos to more complicated issues like nonsensical statements or contradictions in the text. This requires a lot of effort and knowledge, not just of the topic at hand and the language used, but also time to study the work and
its context reduces the risk of defects in the software [2].

The increasing sophistication of software has not just made this process more challenging, but also enabled the development of programs that do have a basic understanding of the context of a program. These analysis tools can either look at the (source-)code alone or study the program while it is being executed. The former is called static analysis and the latter dynamic analysis.

Many issues can be detected by static analysis tools, from formatting and syntactic problems, to crashes when the code is executed later. In order to achieve this, the static analysis tools builds a model (e.g. a call-graph) of the program to be analyzed. Building and working with this models can take a lot of time and take up a lot of space to store information, both growing quickly as the complexity of the analyzed code grows. To still provide insight, less exact algorithms are used to make an analysis feasible at the cost of precision. Many alerts generated by static analysis tools are false positives, i.e. reported errors which are, in fact, not errors. Kremenek et al. [22] report that 30-90% of alerts are false positives. It also has been shown that this is one of the leading causes for developers to reject static analysis tools as it takes too much time to go through the results to find something the developers are actually interested in fixing [19].

1.1 Research Question

This work aims to provide a practical solution to filter already generated alerts in order to improve the usefulness of such tools. It investigates methods of post-processing the list of generated alerts. Our approach is to mine the historic revisions of a software project for information about alerts and meta-data about them in order to build a model that can be used to classify alerts as actionable or not.

Our main question is: Can we improve the usefulness of static analysis tools by filtering alerts based on the software projects historical alert data?

To approach this we break it into following sub-questions:

1. How feasible is it to collect historic data that can be used for model-building?

2. Can we improve the usefulness of static analysis tools with models trained on historic data?

3. Can these models be transferred between sufficiently similar projects?

Gathering enough data to analyze can take a long time and before we address our second question we will run a series of experiments to research
how quickly or slowly this can be achieved. We will recreate parts of an experiment run by Heckman et al. [14] to validate the research and replicate the study on a much larger software project to address the first research question.

Following this we will investigate how we can best use this historic data together with various additional software metrics to reliably predict if an alert is useful or not.

Finally we will test whether we can transfer the model we build with the data on one software project to another one in order to apply it to new projects or use them on software where it is infeasible to gather historic data.

1.2 Thesis Overview

Chapter 2 will give a more in-depth introduction to static analysis tools and especially the ones we will be using: FindBugs, FindSecurityBugs and Static Analysis Tool A\(^1\). It also gives an overview of the state of the art and related work, comment on it and discuss properties of software projects that we use.

In Chapter 3 we will explain the experiments we ran to address the first research question and check the limits of our approach. Their results are discussed as well. The framework we developed, Autobugs, to gather historic alert data will be described, as well as four experiments (1A, 1B, 2A and 2B) we ran using our framework.

The details of building a classifier based on historic alert data will be described in Chapter 4. This entails a description of the chosen features, a brief explanation of the classification algorithms used and concludes with the metrics used to measure the performance of the classifiers.

Our second and third research question will be addressed in Chapter 5 where we will implement multiple machine-learning classifiers using the collected data and evaluate them. On three open-source forum software projects we study the performance of the classifiers. We first train and test the models on each of the projects separately in experiment 3A and then go on to evaluate their cross-project application in experiment 3B. We finish the chapter also testing a custom classifier based on lifetime alone in experiment 3C.

\(^1\)The tool is anonymized because the available license did not grant permission to publish results produced by it.
CHAPTER 1. INTRODUCTION

Future work building on this research and alternative approaches will be discussed in Chapter 6. There, further work will be detailed that we chose not to pursue because of time restraints. It will also describe alternative approaches that we chose not to follow.

Finally, Chapter 7 will give a summary of the results and point out interesting insights. We will also describe the main challenges we faced while working on this topic.
Chapter 2

Background

In this chapter we will describe static analysis and what tools we will be working with in Section 2.1. We go on to discuss some challenges of static analysis and the approaches to solve them by focusing on prior work on post-processing methods in the following Section 2.3. We will shortly describe them and finish by discussing their limitations.

2.1 Static Analysis

Static analysis is a powerful method to find issues in software without executing it. This is especially important as it can show potential problems early in the development, making them cheaper and easier to fix. The types of problems found can range from stylistic conventions on the formatting of the source code, finding security related problems to predicting run-time issues. We will look at the category of tools that focus on the latter as formatting does not change the behavior of a program\textsuperscript{1} and syntax is typically checked by the compiler or interpreter of the language anyway. More interesting are problems that abide the grammar and syntax of a programming language, but cause unexpected behavior when the program is executed.

![Figure 2.1: An alert produced by gcc. This is not an error as the C language allows assignment inside the conditional clause, but it is most likely not intended behavior.](image)

Figure 2.1: An alert produced by gcc. This is not an error as the C language allows assignment inside the conditional clause, but it is most likely not intended behavior.

\textsuperscript{1}Although it can be used to deceive the human reading the code.
and output a list of alerts. An alert is an indication that a check the static analysis tool performs is violated. See Figure 2.1 for an example violation. It shows an alert regarding the assignment of a variable in a conditional clause. Because the comparison of a variable would be done with just an additional '='-character, this is likely to be a typo. Since both expressions are valid, the compiler only outputs a warning but does not abort compilation.

All alert instances of a check share an alert type. Checks can be very simple, such as looking for the use of the insecure function \texttt{gets} in C-code or complex, like finding all possible null-pointer dereferences. While the prior is simple word search on the source code\footnote{To be thorough one should also consider other possibilities of calling the function, of course.}, the latter needs a model of the program at run-time which can take a lot of resources for complex programs.

When an alert is found and reported to the developers, they can either choose to ignore it or react to it. If they decide to act on it, they can either suppress it or fix the underlying problem in the code. If the alert is suppressed, the static analysis tool is instructed not to generate this particular alert or type the next time it is run.

\subsection{FindBugs}
FindBugs\footnote{http://findbugs.sourceforge.net/}\footnote{Correctness, Dodgy code, Security, Performance, Multithreaded correctness, Internationalization, Malicious code vulnerability or Experimental} is an open source static analysis tool for Java and currently in version 3.0.1. It has been originally developed by Bill Pugh and David Hovemeyer. To identify problems, it runs on the compiled Java byte code and uses bug patterns, "a code idiom that is often an error.". Each reported alert is categorized as one of 8 Categories\footnote{Correctness, Dodgy code, Security, Performance, Multithreaded correctness, Internationalization, Malicious code vulnerability or Experimental} and has a confidence and rank. The former indicates how certain FindBugs is that the alert is a real problem and not a false positive, the latter indicates how problematic the underlying error is. The rank is scaled from 20 (least problematic) to 1 (most problematic) and presented to the user as one of the following four tags: scariest, scary, troubling and of concern. Additionally to information about the error, such as the source file containing it, line number and a description, each alert also has a unique \texttt{instanceHash} attribute. It is computed from, among other things, the filename containing the error, the category and type of the error. This is a unique value able to identify a problem even if the surrounding code changes and stays usually the same across many revisions of a software. FindBugs accepts configuration files that allow to in- or exclude source-code based on file-, package-, class- and method-names. They
also allow to filter out alerts based on alert type, priority and confidence.

We chose to include FindBugs in our research as it is easily available as open source project and has been used in similar research over the years.

**FindSecurityBugs**

FindSecurityBugs\[8\] is an open source plugin to the FindBugs tool that extends the available patterns specializing in finding security-related problems. It includes checking for hardcoded passwords, weak encryption algorithms and similar issues and extends the annotation of alerts by a CWE-Nr\(^5\), if applicable. Because security is of particular interest to us and to make the results better comparable to Static Analysis Tool A, we chose to include this plugin.

**2.1.2 Static Analysis Tool A**

Static Analysis Tool A is a commercial static analysis tool for Java and PHP, among other languages. It is focused on web-applications and security errors. The resulting alerts are annotated with a lot of information, such as the error location just like in FindBugs, as well as an equivalent to `instanceHash`. Additionally, it can provide a trace of how an error could be triggered when executing the code, giving the users context to judge if the alert is a true or false positive. Static Analysis Tool A supports similar configuration mechanisms as FindBugs, but we did not encounter a project that configured this tool in any way. We chose this tool to have a state of the art commercial solution included in our analysis and because it was the only commercial tool available to us at the time.

**2.2 Problems with Static Analysis**

In a perfect world we would not need to do anything, as the static analysis tools would generate only true positives and report all problems without missing one. Or rather, not need static analysis because there would never be any mistakes. Unfortunately this is not the case and in order to deal with the number of false positives, which are a big burden on the developers (see [19]), we have several options available:

- Enable the developers to make less mistakes. To not overlook the obvious, it is people who write and deal with code. From training to creating processes for code-review, a lot of actions can improve the quality of code. This paper will not address these topics.

\(^5\)Common Weakness Enumeration, see https://cwe.mitre.org/
CHAPTER 2. BACKGROUND

• Build better static analysis tools. Improving the checks and models that are being run to deal with edge-cases can reduce the number of false positives, although it cannot get rid of them completely. Some issues are ambiguous by design and without additional input a program can not determine if it is a problem or not. This can also mean improving the usability of the static analysis tool, like making changes to the way alerts are presented and including usable explanations of why the alert is considered a problem.

• Improve the output of the static analysis tool. Unlike the previous approach, this is about not changing the tool itself, but rather post-process the generated alerts with additional information that the tool does not take into account. We do take this approach and will introduce related work in Section 2.3.

2.3 Post-Processing Methods

The goal of the post-processing methods is to improve the ratio of useful alerts to the ones developers are not interested in by classifying or ranking the list of generated alerts.

2.3.1 Definitions

First we will give some definitions so we can elaborate on the post-processing methods presented below. These methods can be described with the following properties:

• Classification vs. Ranking. While the former provides a discrete decision if an alert is useful and discards the rest, the latter provides a continuous value expressing the usefulness of the alert. It is possible to make the latter approach produce discrete decisions with threshold-values. This can be useful because ranking does not reduce the number of alerts presented and assumes the users will only look at the topmost results.

• Human- or machine-sourced ground truths. Since most approaches create a statistical or machine learning model, there is a need for training data to build the model. This requires a set of alerts, usually historic ones from the same software project, generated by the same static analysis tool, that are classified beforehand. The alerts can either be explicitly labeled by humans investigating each alert and rating its usefulness or implicitly by an assumed property. As developers need a lot of time to investigate the alerts, it quickly becomes too expensive and most approaches opt for an automated tagging (see Ayewah et al.[3]).
CHAPTER 2. BACKGROUND

- Ranking by individual alert or by a type of alert. Some methods rank or classify each individual alert independently while others operate on sets of alerts that share the same type, e.g., all possible null-pointer dereferences. There are also several approaches that use code-locality (e.g., all alerts in the same file) as a way of grouping.

An important term is the alert-lifetime. It is the time from its first to the last occurrence. It can be counted in revisions, which are the smallest identifiable changes to the source-code or actual time between when the revisions have been checked into the software project. Multiple approaches work with the lifetime because it can be automatically obtained from the source code repository if one is available for a given project. The lifetime of an alert is an interesting property as it can be assumed that a short lifetime means the alert is found and important enough to fix quickly[20].

Another ubiquitous term in the literature we reviewed is the actionable alert. According to Ruthruff et al.[27], actionable alerts are “legitimate warnings [true positives] that will be acted on by developers”. Heckman et al.[13] describe an alert being actionable, “if a developer determines the alert is an important, fixable anomaly”[13], where anomaly is a reference to the IEEE definition as a “condition that deviates from expectations based on requirements specifications, design documents, user documents, or standards, or from someone’s perceptions or experiences”.

An actionable alert is a more practical definition as it is not just a true positive. While a true positive is an alert that reports an actual problem in the code, the actionable alert is one that will be acted upon. Not all problems are important enough to be acted upon because each change to the software might introduce new anomalies. If a bug hasn’t been found for a long time and does not cause problems in the software, it might be too risky to change the code to fix it[19].

Because getting the classification of alerts by developer feedback is not suitable, Heckman et al.[13] use an automated approach to classify the alerts: Each alert that is still open in the latest revision is un-actionable. All others, that disappear before the latest revision, are actionable, unless the file containing the alert disappeared at the same time as the alert. In this case, it is argued, no assumption can be made about the alert and it is discarded from further use. Figure 2.2 shows the three examples, where alert 1 is actionable, alert 2 discarded and alert 3 un-actionable.

This definition however, is not universal. Allier et al.[1] have defined actionable alerts the same way, i.e. all alerts that disappear before the latest revision, but not due to deletion of the containing file. The definition of discarded and un-actionable is swapped: All alerts that are open in the latest revision, it is argued, can not be classified as (un-)actionable and they are discarded. The un-actionable alerts are all that disappeared due to file-
CHAPTER 2. BACKGROUND

Figure 2.2: A fictional repository with six revisions. Alert 1 is actionable, because it disappears after revision 3. Alert 2 disappears too, but the file containing it gets deleted so we discard it according to Heckman et al. and ignore it. Alert 3 is still open in the latest revision and thus un-actionable.

deletion as they were never addressed before the file got completely removed.

In this work we use the definition of Heckman et al. as illustrated in Figure 2.2 to automatically generate a ground truth. We chose it because it seems in our opinion better to not assume properties about deleted files as we do not investigate why they have been removed.

2.3.2 Previous Work on Post-Processing

Comparing the performance of post-processing methods is not easy, because the research of this topic uses varying projects and tools. The results can vary due to the analyzed project, the static analysis tool or the method itself. Allier et al.[1] re-implemented a number of methods to run on the same software projects and with the same static analysis tools to compare multiple approaches in a structured way.

These are the methods investigated by Allier et al. that rank individual alerts:

- **FeedBackRank**[22] works with the assumption that “alerts sharing an artifact location (method, class, package) tend to be either all actionable or all un-actionable”. It requires the feedback of developers marking alerts as actionable or un-actionable and then uses a Bayesian Network to update the ranking of all other alerts for the system.

- **AWARE**[16] considers the success rate of the alert type, that is, how many alerts of the same type have been marked as real problems by
developers. Additionally, it uses code locality of the individual alert to influence the ranking of nearby alerts. The assumption, very similar to FeedBackRank, is that faults flock together and if a potential problem is found among many confirmed faults, it is more likely to be important itself.

- RPM[27] is using a logistic regression model to classify individual alerts. Because they are considering a large number of properties of alerts to build the model, the method includes a step to reduce this number by calculating how much each property actually contributes and discarding all below a certain threshold. The ground truth is human sourced and the model tested in a large-scale experiment conducted at Google over multiple months in which developers rated individual alerts.

The following methods investigated by Allier work on alert types, ranking sets of alerts and not individual ones:

- AlertLifeTime[20] first calculates the lifetime per alert type and prioritizes types with a low value. It assumes that alerts which are fixed quickly, are more important to the developers than ones that take a long time to be fixed. The edge-case of alerts still open in the latest revision are given a penalty when calculating the lifetime as they are assumed to be not as important since they are still open.

- EFindBugs[28] stands for Effective error ranking for FindBugs. Periodically, the ratio of actionable and un-actionable alerts is calculated per alert type and used to rank those. The classification of actionability is done manually by developers on historic alerts or imported from another project.

- z-Ranking[21] ranks alert types with a quite distinct approach, as it not only needs the number of alerts generated per type, but also the number of times a check was performed but no alert generated. That usually means that the static analysis tool needs to be modified, because every time a check is performed, an output needs to be generated. Usually a static analysis tool only reports when a check fails as alerts. The ratio between successful checks and unsuccessful ones is then used to rank the types. This is the only method that does not rely on either a manual or automatically generated ground truth.

Heckman et al.[15] created a systematic literature review summarizing many methods and their properties. Because the original publications did not use the same software projects and tools they did only a limited comparison.
A very recent survey by Muske et al.\cite{23} mentions additional approaches, not just ranking. It includes clustering, methods combining static and dynamic analysis as well as tools to simplify inspection of the alerts. Building lightweight static analysis tools is also discussed as an alternative to minimize false positives.

### 2.3.3 Limits of Previous Work

Common problems of the mentioned approaches are that it is difficult to get the required properties of alerts. One of the most difficult to obtain is the ground truth of true positive alerts. Many methods assume there exists already a number of classified alerts, which can be used to build models and find other properties to predict the actionability.

This ground truth is difficult to obtain, because it needs human interaction - if there was an algorithm to identify true positives, this research would not be necessary. Having alerts tagged by developers costs lots of resources and so many researchers opt for another option. See \cite{3} for an exception to this. Secondly, developers might disagree on what exactly are true positives. This can be tied to the practices of the project and personal preferences. The issue is not just about the code, but also the context of the production. Is it better to change existing code that has been running for a long time to address a potential issue that has not caused problems in practice? Or is it better to "not fix a working system"\footnote{Paraphrasing "If it ain’t broken, don’t fix it".}

Substituting the ground truth by automatically generateable data (e.g. the actionable alert in Section 2.3.1) might seem like a convenient replacement, but brings problems as well. Models that rely on historic data are heavily biased towards the developers behaviors and do not reflect good practices. In a simple example, let us consider a project that has very many SQL-injection flaws. They are reasonably easy for static analysis to spot and many programming languages provide constructs to avoid them without much complication. But if the developers do not know what an SQL-injection is, they never fix those issues. These alerts will have a very long lifetime or might still be unfixed at the most recent revision. A model build solely on the historic data would have no way to distinguish a type of alerts that the developers are genuinely not interested in and those they mis-identify as non-problems. For more on this, see Chapter 6.

Another problem is that there simply is not enough data to be mined, eg. when a project does not have enough revisions to warrant that the historic data is representative and the resulting model significant enough. This also
means that it is difficult to generalize results as the model can vary greatly across projects.
Chapter 3

Framework for Collecting Historic Alert Data

In this chapter we introduce Autobugs, a framework to collect historic alert data on interchangeable software projects and static analysis tools. The goal is to answer research question 1, how feasible it is to collect historic data that can be used for model-building, and find the limiting factors for this approach. Further, we investigate the different properties of the analyzed software projects, JDOM and Elasticsearch.

First we address the goals, the workflow and decisions regarding the setup of our framework in Section 3.1 and explain our reasoning behind them. In experiment 1A we recreated part of the work done by Heckman et al.[14] by collecting historic alert data on the JDOM project to compare our results with theirs.

We go on to exchange the software project used (JDOM) by a much larger one (Elasticsearch) in experiment 1B to compare the impact of size and maturity.

Then, in experiment 2A and 2B, we use a different static analysis tool and run it on the same projects as in experiment 1A and 1B to compare our results across different tools.

In all experiments we calculate the lifetime of alerts and their classification as (un-)actionable according to Heckman et al. as explained in Section 2.3.1. Additionally, we investigate possible relations between the alerts generated and the size of the projects.

We conclude this chapter with a discussion of the results concerning the feasibility of collection of historic alert data and the insights and problems we had while building Autobugs and executing the experiments.
3.1 Description of Framework Architecture

We developed a framework, which we called Autobugs, in order to automatically run static analysis tools on past revisions of a software project and save the results for further analysis. It is designed to easily exchange the static analysis tool and software project to inspect. The generated data can be used to find correlations between alert properties and also as training or test data for building models that classify the actionability of an alert, as described in Section 2.3.1.

We modeled our approach after the experiments conducted by Heckman et al.[14] so we could compare our results with the numbers they published. In their experiment, Heckman et al. ran the static analysis tool FindBugs, (see Section 2.1.1) on multiple revisions of two software projects. These projects used in the original work are JDOM[18] and the runtime of the Eclipse integrated development environment\(^1\). Both projects are open source Java programs. JDOM is a smaller library for handling XML data with 16k lines of code, while the Eclipse runtime is a bigger project from the software development domain with 600k lines of code. Heckman et al. then used this data for creating and evaluating machine learning models to classify alerts as actionable or un-actionable. In our experiments 1 and 2, we only replicated the gathering of alert data and not the model-building. We also chose to pick only JDOM to use in our first experiment as its small size enabled us to quickly iterate our development of the framework by minimizing the time it takes for a complete run.

Our framework is run on a dual core i5-4250U CPU @ 1.30GHz stock laptop with 8GB RAM.

3.1.1 Workflow

A complete run of our framework consists of 4 steps:

1. If it is the first run, initializing the result database and creating a list of all revisions that is numbered with 1 being the oldest.

2. Then, for every \( n \)-th revision, checking that revision out of the software repository and running the static analysis tool, if necessary building the project first. The tool’s output is saved to a file identified by the project’s name, the tool’s name and the revision. In case a build fails, we note this but ignore the commit in the analysis.

3. After all revisions that were selected have been analyzed, we go over all resulting files and parse them to extract the alert information into

\(^1\)http://www.eclipse.org/rt/
CHAPTER 3. FRAMEWORK FOR COLLECTING HISTORIC ALERT DATA

4. Finally, the alerts are processed to calculate resulting properties, like their lifetime.

In order to add a static analysis tool, only an abstract function has to be implemented that is called with the path to the target, the revision and possibly other options to the tool.

The process to add a project is more involved, since the location of the source and properties can change throughout the history of the project. We only consider the source code that does not include automated tests or code to manage the installation or deployment of the software. For the first run, we let the analysis start at the most recent commit and set the target folder and possible build and tool properties. If the tool requires the program to be build, we define build and clean commands that are run before and respectively after the analysis. In case any of the steps of building, analyzing or cleaning the project fails, the framework throws an error and stops. We then investigate the cause of the error and adjust the build-, analysis- or clean-commands accordingly for the failed revision and continue with this until the first commit is reached or we have analyzed enough revisions.

3.1.2 Selecting a Subset of Revisions

All projects we analyze use the version control system git[12]. It allows to have multiple so-called branches of parallel development lines, which make it tricky to navigate chronologically. Tracking alerts becomes a problem when a new development branch is forked and then later merged back into the master branch, effectively providing two different histories. Heckman et al. mentioned the project they investigated used SVN\(^2\) and the now discontinued version control system CVS\(^3\). They did not go into how they handled branches and we decided to create a revision history by going backwards from the most current revision in the master branch and always picking the first parent, which is in our case always on the same branch (the master branch). We then numbered the resulting series of commits starting with 1 at the oldest commit and counting upwards. In this step we did not skip any revisions in order to avoid problems further down the road in case we wanted to change something.

We then access every \( n \)-th revision through git and run the static analysis tool on it, compiling the project first if it is to be analyzed by FindBugs. The result is a file whose name is composed of the names of the project, the

\(^2\)https://subversion.apache.org/, formerly http://subversion.tigris.org/
\(^3\)https://en.wikipedia.org/wiki/Concurrent_Versions_System
tool used and the revision hash in order to be identifiable for later use. This hash is a unique value identifying a revision within git.

### 3.1.3 Computing Lifetime & Actionable Alert Classification

The static analysis tools we use helpfully provide an instance-hash property for each alert, which we can use to track the same alert over multiple revisions. It is computed as the hash of other alert properties, like the name of the containing file and the method name with its signature.

When computing the lifetimes of alerts (see also Section 2.3.1), we only consider the first and last occurrence of an alert and ignore if it temporarily disappears, just as Heckman et al. Unfortunately they do not give a reason for this decision. We chose to ignore the temporary disappearance because we do not investigate the cause for it. However, as the alert is the same based on its identifier provided by the static analysis tools, we can assume that the reason for the alert disappearing was not a complete fix.

The classification of (un-)actionability is then built with the lifetimes. First, all alerts that are open in the latest analyzed revision are classified as un-actionable. Then, for every remaining alert it is checked if the file containing it disappeared together with the alert. If it is still present after the alert disappeared, it is classified as actionable, else it is classified as deleted and ignored for all further purposes.

Because both FindBugs and Static Analysis Tool A are using the filename as part of their instance-hash, refactoring the code by changing filenames results in different instance-hashes and creates uncertainty in the resulting
Another issue is the branching of git. If changes are made on a branch, we are not noticing them until they are merged into the master branch, as shown in Figure 3.1.

Both these issues are visible in our experiments as discussed later in this chapter.

3.1.4 Configuration Files of Static Analysis Tools

Each tool can be customized to in- or exclude certain directories, files or even classes and methods as well as sets of alerts sharing a property like type or confidence. In JDOM we found such configuration files for FindBugs. For the latest revisions, they contained directives on where the source code was located and to explicitly ignore test-files. As we were doing this anyway, we chose to have our Framework ignore such files.

3.2 Experiment 1A - FindBugs with JDOM

In this experiment we try to build and, if building succeeds, analyze every 16th revision of the JDOM project with the static analysis tool FindBugs and save the results. The number of revisions to skip is chosen as it is low enough to enable a very quick collection of data while resulting in more total revisions being analyzed. We proceed to calculate lifetimes and classify the alerts as (un-)actionable to compare our results with the work of Heckman et al.

3.2.1 JDOM

JDOM is an open-source Java library to input, process and output XML data that we chose to analyze with the FindBugs tool (which is addressed in Section 2.1.1) to compare our results directly with the work of Heckman et al.[14]. It has been developed since 2000 and its current release is version 2.0.6. It consists of around 16,000 lines of Java code, as calculated by the cloc tool\(^4\), excluding test-code.

3.2.2 Comparison to the Work of Heckman et al.

Heckman et al. sampled 48 of the available 1165 revisions at that time, analyzing every 24th, in one hour and 23 minutes. We managed to sample twice the amount in half the time, analyzing 101 revisions of 1609 available, analyzing every 16th, in 39 minutes for a complete run as described in

\(^4\)https://github.com/AlDanial/cloc
Section 3.1.1. Unfortunately the original work did not mention the specification of the setup to compare them.

To our surprise we were able to build a much higher percentage of revisions successfully by re-configuring the build parameters every time a build failed. This was a manual process done in the first run after which our framework changed parameters accordingly to build the project. We build and analyzed 90 out of the 101 sampled revisions while Heckman build and analyzed 29 of the 48 sampled revisions.

We identified 198 individual alerts in total, with 119 being actionable, 69 deleted and only 10 un-actionable. The total number of alerts is less than half the number of alerts reported in the original experiment, with a very different distribution of numbers in the categories. If we consider only the first 1165 revisions, the total number of alerts found shrinks to 83, not even a fifth of what was reported in the original research.

3.2.3 Analysis and Visualization of the Data

Let us begin by looking at the number of alerts for each analyzed revision. Figure 3.2 shows the evolution of the number of reported alerts over time, first by commit-number and then by commit-date. We plot both versions as they show different patterns more clearly.

In Figure 3.2a we can see the number of alerts per commit revision. Note that we did not analyze every single revision but only every 16th. Some revisions are not shown, e.g. between the last and next-to-last plotted data point. In these cases we failed to successfully build the project and skipped analysis.

While we can see the overall rise and fall of number of alerts in the first figure, we are missing a scale of time, which is shown more clearly in Figure 3.2b. Here the number of alerts is plotted against the actual date of the commit we analyzed. We can see intense periods of work in 2001 and 2011 and that between 2005 and 2011 very little was committed compared to the aforementioned times, when the project was more in maintenance mode than being actively developed.

We marked four interesting points in Figure 3.2 with vertical lines, where we see the number of reported alerts per revision.

1. First is the latest revision that was used by Heckman et al. As they provided the exact number of revisions they had available, which is 1165, we know that their research was conducted at the end of a long period of low activity on the project.

2. Next is revision 1178, in which the folder structure was renamed from jdom to jdom2. At this point a new developer took over and started refactoring large portions of the code to prepare for a new version of the project.
Figure 3.2: The number of alerts generated by FindBugs for JDOM revisions

(a) by revision number

(b) by commit-date
3. Revision 1261 marks the first time in which FindBugs configuration files were introduced in the source code versioning system git. We assume it was then used to analyze the code but we could not find an indication of how often it was run or if it was required to be run at all. It is possible that FindBugs was used before then although we could not find evidence of it. A sharp decline in the number of bugs reported in the later revisions is very visible, most likely because FindBugs was regularly used then.

4. Finally, revision 1520 marks the official release of JDOM2, a major update to the project. Surprisingly, we managed to successfully build less versions from this time than before. We suspect that most revisions were not meant to be releases, but only small changes that were bundled into a release less often.

Problems With Tracking Alerts

In Figure 3.3 we visualize the lifetimes of alerts. Each line represents an actionable alert and goes from the earliest to the latest revision it was observed.

We can see a peculiar pattern at mark number 2, at revision 1178: a lot of new actionable alerts pop up. Because the project was just re-factored from jdom to jdom2 in this revision, these alerts are not new. No code has been changed, just the paths to the source code changed. FindBugs does not recognize them as the same alerts that have been around before and Heckman et al. do not mention it in [14], as this happened after their study.

Since the project is also given to a new main developer, it could even be regarded as a separate project than the continuation of the same. The underlying problem however is the way alerts are tracked across revisions: Because a change in the file name gives an alert a completely different identifier, it is not possible to recognize simple refactoring. This influences the alert lifetimes, as alerts that are the same before and after refactoring are identified as two different ones by the static analysis tool. The first of these two alerts is classified as deleted and discarded from analysis and further adds uncertainty to the data. See chapter 6 for more on tracking alerts between revisions.

3.2.4 Discussion

Our replication of the experiment by Heckman et al. managed to generate a lot fewer alerts. We suppose this is due to the advance of the FindBugs static analysis tool. Just like JDOM, it has seen a major version release and we did not run our experiment with an old version but with the newest one.
CHAPTER 3. FRAMEWORK FOR COLLECTING HISTORIC ALERT DATA

Figure 3.3: The lifetime of actionable alerts generated by FindBugs for JDOM revisions

(a) by revision number

(b) by commit-date
CHAPTER 3. FRAMEWORK FOR COLLECTING HISTORIC ALERT DATA

This experiment also showed that changing versions can be a problem - not just of the analysis tool, but the analysis target as well. As explained by Bessey et al.[5], improving a static analysis tool can mean increasing the number of generated alerts because the tool implements more checks. In our case we generated a lot less alerts on the same project with more analyzed revisions. We suspect that FindBugs became better at reducing the number of reported alerts and at filtering out the false positives.

We chose not to look into the distribution of alert types or use the alert data in further experiments of Chapter 4, because of the very small sample size. It seems generalizations from just a couple hundred alerts are very weak, but the visualization of the alert-lifetimes was very helpful in identifying a problem with our approach.

3.3 Experiment 1B - FindBugs with Elasticsearch

In this experiment we analyze every 32nd revision of the Elasticsearch project with the FindBugs static analysis tool. We limited our analysis to the most recent revisions and stopped after analyzing 103 revisions. We proceed, as in experiment 1A, to calculate lifetimes and classify alerts as (un-) actionable to compare the project with JDOM.

3.3.1 Elasticsearch

In this experiment we switched JDOM for a much larger project, Elasticsearch [7], an open-source search engine written in Java. With over 280,000 lines of code it is about an order of magnitude bigger than JDOM. We chose it because the work of Beller et al.[4] made us believe that developers were required to analyze their code contribution with FindBugs. Further investigation showed however, that this is not the case. Beller et al. did state that the use of a static analysis tool was obligatory and that FindBugs was used, among other static analysis tools. However, looking at the repository and the forums where bug reports are handled, as well as Elasticsearch’s contribution guidelines we found that a style checker was mandated before submitting code, but FindBugs was used only occasionally. We found mentions in the commit messages and the issue-tracker of Elasticsearch\(^5\).

Unfortunately the project does not mandate the use of the static analysis tool that reports bugs.

\(^5\)https://github.com/elastic/elasticsearch/issues?q=findbugs
3.3.2 Analysis and Visualization

Because of its size and thus increased runtime, we did not analyze all revisions of Elasticsearch but instead limited our framework to analyze every 32nd of the most recent revisions, limiting it to a total of 103. We chose this number to be able to compare the runtime with JDOM and to limit the analysis time for Elasticsearch. A complete run would have taken almost 4 days, still skipping 32 revisions and over a week of continuous runtime for experiment 2B, which was not feasible.

The different shape of the number of alerts in Figure 3.4 shows a stark contrast to JDOM seen in Figure 3.2. The number of bugs is much more stable for Elasticsearch. Some peculiar moments can be observed however:

1. The first mark, which is at revision 10689. Around this time, multiple language bindings have been added to the project at once from previously independent repositories.

2. The next marker at revision 11649 and 11681 indicate a large gap, not evident in Figure 3.4a but in Figure 3.4b. For more than half a year we have data missing, which we cannot simply attribute to skipping 32 revisions between analysis as elasticsearch has seen multiple releases during that time.

3. Mark 3 indicates the first revision that we analyzed after the almost 6-months period and when development on the master branch resumed.

Artifacts Caused by Git

We investigated why there is such a long gap in our data during last year and found the reason to be tied to the version control software git (See Section 3.1.3). It allows developers to work on multiple branches simultaneously before merging their progress, which it also saves in the commit history. When building a single path through the history, we chose to always pick the first parent of a commit as this guarantees to stay on a single branch. However, it seems the elasticsearch developers did not commit to the same branch for a long period of time for unknown reasons, but instead we could see over 50 different branches being worked on during that time. This resulted in our missing revisions.

This is even more evident in Figure 3.5b, where we can see a lot of alerts simultaneously disappearing after the gap in our data in July 2016, but it is highly unlikely that they were not fixed earlier in a different branch and we just don’t have the lifetime on the correct branch for them. On the one hand this skews the lifetimes to be longer and not representing when the
Figure 3.4: The number of generated alerts for each analyzed revision of Elasticsearch by commit nr (above) and by commit-date (below).
Figure 3.5: The lifetime of actionable alerts in the elasticsearch project by commit nr (above) and commit-date (below).
alert was fixed. On the other hand we can assume that if an alert has been fixed on a different branch than the master, it is not important enough to be merged quickly into the master branch.

3.4 Experiment 2A and 2B - Static Analysis Tool A with JDOM and Elasticsearch

In experiment 2A and 2B we exchange the static analysis tool FindBugs used in experiment 1A and 1B for Static Analysis Tool A. Analog to the first two experiments we run it on JDOM and Elasticsearch, analyzing the same revisions and calculate alert lifetimes and classification.

3.4.1 Static Analysis Tool A

In our second set of experiments we switched the static analysis tool to Static Analysis Tool A (see Section 2.1.2). As it does not require compilation of the code to be analyzed, we can skip this step. Otherwise the procedure is the same as in experiments 1A and 1B:

First the history is built, skipping the same number of revisions as with FindBugs to limit the time it takes to run a complete analysis. Then the selected revisions are analyzed by the tool and the results saved. In the last step a database is filled with the alert data that is used to generate alert lifetimes and classes.

As we can see in both Figure 3.6 and Figure 3.7, Static Analysis Tool A is producing a much larger number of alerts.

<table>
<thead>
<tr>
<th></th>
<th>JDOM</th>
<th>Static Analysis Tool A</th>
<th>Elasticsearch</th>
<th>FindBugs</th>
<th>Static Analysis Tool A</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>198</td>
<td>749</td>
<td>total</td>
<td>1905</td>
<td>13692</td>
</tr>
<tr>
<td>AA</td>
<td>119</td>
<td>388</td>
<td>AA</td>
<td>413</td>
<td>4116</td>
</tr>
<tr>
<td>UA</td>
<td>10</td>
<td>163</td>
<td>UA</td>
<td>672</td>
<td>4676</td>
</tr>
<tr>
<td>DEL</td>
<td>69</td>
<td>198</td>
<td>DEL</td>
<td>820</td>
<td>4900</td>
</tr>
</tbody>
</table>

Table 3.1: Overview over number of unique alerts generated by the software project & static analysis tool combination. AA means actionable alert, UA un-actionable and DEL signifies the number of alerts that vanished due to file deletion.
Figure 3.6: The lifetime of alerts in the JDOM project by commit nr (above) and commit-date (below).

(a) by revision number

(b) by commit-date
Figure 3.7: The lifetime of alerts in the Elasticsearch project by commit nr (above) and commit-date (below).
CHAPTER 3. FRAMEWORK FOR COLLECTING HISTORIC ALERT DATA

JDOM

For JDOM, the number of alerts are evolving similar up until the FindBugs configuration-files are introduced to the repository, as seen in Figure 3.6a at mark 3 (revision 1261). While the number of alerts generated by FindBugs declines afterwards, the number of alerts generated by Static Analysis Tool A does not follow this decreasing trend. It seems that after the introduction of FindBugs to the project, its alerts were explicitly addressed and fixed and that those alerts were not strongly related to the alerts generated by Static Analysis Tool A. If they were, we would expect to see an ongoing similarity between the number of alerts.

Elasticsearch

Just like in experiments 1A and 1B we can see artifacts from the development process.

In Figure 3.7b, the second mark at revision 11649 shows the beginning of a long timespan that is missing. As mentioned in Section 3.3.2, investigating this showed that much development work was done on a separate branch. Because of how we handle creating a time-line of revisions to analyze (see Section 3.1.2), this work is not visible to us.

Another interesting moment is at mark 1, revision 10689, when large parts of external code are added to the project as bindings for other programming languages. The merging of the code takes less than 32 revisions, which is our resolution for the data. The merging causes in a jump of alerts for FindBugs, but did not change as significantly the number of alerts reported by Static Analysis Tool A. We could not find an explanation for this other than the differences in analysis techniques used by the different tools.

3.4.2 Lifetimes

JDOM

Similar to the Alert lifetimes graphs (Figures 3.3 and 3.5) in Section 3.2.3, Figure 3.9a shows a big number of alerts appearing at revision 1178 when the project was re-factored. Because Static Analysis Tool A uses a technique to track alerts across revisions that, like FindBugs, depends on the filename, the change in the folder name containing the source code is clearly visible as well. Other than this it is similar in shape to the FindBugs graph in Figure 3.3, with the exception of producing an overall greater number of alerts.

Elasticsearch

As with FindBugs in Figure 3.5b, artifacts of the development process are very clearly visible in Figure 3.9b. The sudden increase of alerts and sudden
Figure 3.8: The lifetime of alerts in the JDOM project by commit nr (above) and commit-date (below).
disappearance of many alerts simultaneously can be attributed to development taking place in a different branch. Because branches in git are invisible to our method, we can not pinpoint when exactly an alert was first created nor when it exactly disappeared, until the development branch was merged back to the master branch.

It took just about three and a half days to generate the historic alert data for only 103 revisions. A full run not skipping any revisions would take over a year with our setup. This shows how infeasible it is to collect historic data if it is not already available for such a large project.
Figure 3.9: The lifetime of alerts in the Elasticsearch project by commit nr (above) and commit-date (below).
### 3.5 Discussion of the Experiments

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Project</th>
<th>Tool</th>
<th>Time for Analysis</th>
<th>Revisions Sampled</th>
<th>Alerts Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heckman 2009[14]</td>
<td>JDOM</td>
<td>FindBugs</td>
<td>1h 23m</td>
<td>48</td>
<td>420</td>
</tr>
<tr>
<td>1A</td>
<td>JDOM</td>
<td>FindBugs</td>
<td>39m</td>
<td>101</td>
<td>198</td>
</tr>
<tr>
<td>1B</td>
<td>Elastic-Search</td>
<td>FindBugs</td>
<td>9h 59m</td>
<td>103</td>
<td>1905</td>
</tr>
<tr>
<td>2A</td>
<td>JDOM</td>
<td>Static Analysis Tool A</td>
<td>5h 56m</td>
<td>101</td>
<td>749</td>
</tr>
<tr>
<td>2B</td>
<td>Elastic-search</td>
<td>Static Analysis Tool A</td>
<td>3d 12h 36m</td>
<td>103</td>
<td>13692</td>
</tr>
</tbody>
</table>

Table 3.2: Time to generate alerts for given number of revisions on a dual core i5-4250U CPU @ 1.30GHz with 8GB RAM

Addressing our first research question, we found that analyzing every revision of a project such as Elasticsearch, which has about 600k lines of code, is already infeasible. We can reduce the time taken by skipping revisions and possibly miss very short-lived alerts that fall between our resolution.

We also see that the usage of a static analysis tool has an impact on the amount of alerts produced by the tool, as evident in experiment 1A. It would be most beneficial to study a project that is aware of each alert generated and fixes all alerts it deems important enough to have, but unfortunately we were not able to find such a project.

Furthermore, it has come to our attention just how few revisions were generated by the original research of Heckman et al.[14] With just 29 that compiled and were analyzed. It is very difficult to generalize results with so few data points. A balance needs to be found in order to enable reasonable quick generation of historic alert data and analysis of a sufficient number of revisions.

#### 3.5.1 Actionable Alert Metric

The actionable alert classification based on lifetime is very good at automatically generating data for any project with a revision history and on which
an ASAT can be run automatically. This effectively lets the programs run completely unsupervised, generating its ground truth automatically. But at the same time it introduces an uncertainty why a particular alert disappeared between revisions, because it is highly dependent on the ability to track alerts between revisions. A simple renaming breaks both FindBugs and Static Analysis Tool A’s built-in alert identifier and we decided to not look further into the this as it presents a different challenge than the one we were investigating. The actionable alert classification further assumes that any change, which made an alert disappear, was made with the intention of addressing the underlying problem of the alert. This might not always be the case and needs to be taken with a grain of salt.

### 3.5.2 Alert Lifetimes

As mentioned in Section 3.1.3, the lifetimes of alerts are not completely accurate because they are tracked along the master-branch only. We can not track the exact commit when an alert was first generated by only looking at the master branch and the same goes for the commit that makes the alert disappear. Many alerts appear and disappear at the same time because of branching and merging of commits. In order to track the lifetime more accurately, more branches would need to be considered and a more complex definition for the alert lifetime needs to be found, as each branch possibly results in a different lifetime for an alert. More on this can be found in Chapter 6.
Chapter 4

Feature Extraction and Classifiers

In this chapter we will explain the process of training and testing the classifiers, which will be evaluated in Chapter 5. We will give an explanation of the overall process in 4.1, introduce the features in Section 4.2 and the classifiers in Section 4.3. In the final Section 4.4 we will explain the chosen metrics, which will be used to measure the performance of the classifiers.

4.1 Process of Building a Classification Model

![Diagram of the process of training and evaluating models of actionable alert classification.](image)

Figure 4.1: The process of training and evaluating models of actionable alert classification.

The main steps necessary to building a classifier based on historic alert data are outlined in Figure 4.1. It begins with gathering the raw data. In our case this is done by the Autobugs framework, generating alerts for a
subset of past revisions. As we have shown, it is not feasible to generate the historic data for every revision of a project of about 600k lines of code.

The next step is feature extraction. The features we will be using in our evaluation of classifiers are described in Section 4.2. This transforms the data into a format that can be used by the classifier.

Afterwards, the classifier is trained on the data provided and generates a model, an internal representation of the data it has been trained on. This varies from classifier to classifier and is described in Section 4.3. The model is an instance of the classifier already trained on samples and ready to classify new samples.

Finally, the model can be used to classify a new sample. To evaluate the classification we use the accuracy, precision and recall measures, described in Section 4.4.

4.2 Features

Features are characteristics of an alert. With the historic alert data gathered in Chapter 3, we already have some features that we will be using: The lifetime, type, confidence and severity of an alert. Additionally, we add some more as calculated by the Software Analysis Toolkit (SAT) developed by SIG. The SAT is used to assess software quality and support consultants in evaluating metrics such as complexity, maintainability and software quality. It does so by measuring various properties of software. We selected the additional features reported by the SAT because we hypothesized they will be an indicator of actionability.

- Lifetime in seconds based on the commit-time. We chose time instead of commit-numbers because the time between commits can vary, making it incomparable otherwise.
- The type of the alert by Static Analysis Tool A. We use the most precise type available. We speculated that certain types might be more important to the developers and thus an indication of actionability.
- Confidence, as reported by Static Analysis Tool A. This is an indicator of how certain the tool is that the alert is a true positive.
- Severity, as reported by Static Analysis Tool A. This is an indicator of how important the alert is according to the tool.
- Lines of code in the containing unit as reported by the SAT. This is typically a method or function. We speculate that a longer and thus more complicated unit might be less actionable, because it does require more effort to recognize and fix errors.
CHAPTER 4. FEATURE EXTRACTION AND CLASSIFIERS

<table>
<thead>
<tr>
<th>sample</th>
<th>type</th>
<th>type A</th>
<th>type B</th>
<th>type C</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample1</td>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sample2</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sample3</td>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sample4</td>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: Example of one-hot-encoding: The 3 different values for the type-feature become 3 features with only one set to 1, the others to 0.

- Cyclomatic Complexity or McCabe-value indicates the number of independent paths through the unit, as computed by the SAT. Same as the lines of code, it might show a relation to actionability.

- The containing unit’s fan-in value, which counts how many other units are calling it. It might be that a unit being called very often would have alerts resolved more quickly, because misbehavior would be detected more quickly.

- The containing unit’s fan-out value, which counts the calls made from the unit containing the alert to others. Another measure of complexity we wanted to include.

The alerts were automatically labeled based on their lifetime using the method described in Section 3.1.3 as (un-)actionable in order to enable supervised learning. All alerts that are open in the latest revision are classified un-actionable. All alerts that disappeared before, but not because the containing file was deleted, are actionable. This means the target classes are defined in the learning data and do not have to be estimated by the classifier, the learning is done supervised.

All features except the type are ordinal values, that means that the ordering of the number is meaningful. The classifiers we use work best on those ordinal values. However, the type is not ordinal but categorical. One can compare the height of trees because it is an ordinal value, but ordering the type (eg. ash, birch or maple) would not make much sense. Representing each type with a number is not meaningful for the classifier as the ordering of those numbers is arbitrary. In order to use the type, we apply a technique called one-hot-encoding. For each type, we make a new, separate feature. This feature is set to 1 for the type of the given alert and to zero for all other type-features, as demonstrated in Table ???. Only one of those features is encoded as ”hot”, thus the name.
4.3 Classifiers

We decided to use machine learning methods to build a model from the historic data. To compare how well it works, we chose three different classifiers, a decision tree, a linear support vector machine and a non-linear support vector machine, and added a random classifier as a baseline. While the last one does not need training, the first three need to be first trained with data to build a model and then can classify new alerts based on the trained data and their parameters.

For the implementation of our classifiers we used scikit-learn[25], a Python library that implements various machine learning techniques and related functions. It provides an easy-to-use interface to our existing data-gathering framework.

Decision Tree

The decision tree, as the name suggests, builds a tree that is similar to a flow chart. A new alert follows a series of yes/no questions about its features before arriving at a leaf node and being classified. The tree is built by starting with the whole set of samples and then determining which feature is best at divining the set into the two classes. This division is optimized by minimizing the gini-impurity, a measure of the chance of misclassifying a random alert.

We chose the decision tree because it enables us to have a human-readable explanation of a classification and decided to use the default parameters. This means there is no upper limit on the number of leaf nodes or the maximum depth of the tree. This will most likely result in over-fitting the training data (see Section 4.4.1) but will give us a complete explanation of the reasoning behind the classification.

Linear and Non-Linear Support Vector Machine

The linear support vector machine (LSVM) is recommended by scikit-learn[1], for our problem of classifying labeled data with less than 100k samples. It uses a D-dimensional space, where D is the number of features of an alert. In this space it tries to find a hyperplane that best separates the actionable from the un-actionable alerts. Having found this plane, new alerts to be classified are then placed in the hyperspace and it is computed on which side of the hyperplane it is, which results in the class for the new alert. If the data is not linearly separable, we use a non-linear support vector machine (SVM) that works on the same principle but uses the so-called kernel-trick to increase the dimensionality of the data until it is separable by a hyperplane. We used default parameters for both the linear and non-linear SVM.

CHAPTER 4. FEATURE EXTRACTION AND CLASSIFIERS

Random Classifier
The random classifier adds a baseline that just randomly classifies each alert as either actionable or un-actionable. It is not provided by scikit-learn but implemented ourselves. It is not meant as a real classifier but to verify that the results are actually better than just randomly assigning classes with a probability of 0.5.

4.4 Performance Metrics
In order to evaluate the classifiers we use the different metrics that are computed from the result of a classification and the expected outcome. For the following definitions, an actionable alert that is recognized as such by the classifier is a true positive and an un-actionable alert classified as such is a true negative. A false positive is an un-actionable alert classified as actionable.

- **Accuracy** is \( \frac{\text{TruePositives} + \text{TrueNegatives}}{\text{TotalNumberofSamples}} \). It gives an overall indication of how many alerts are classified correctly by the model according to their label.

- **Precision** is \( \frac{\text{TruePositives}}{\text{TruePositives} + \text{FalsePositives}} \). This is something we are particularly interested in as a high precision indicates a low false positive rate among the result. However, precision alone is not meaningful because if only a single sample was classified correctly as actionable, it would be still 1 - the best value.

- **Recall** is \( \frac{\text{TruePositives}}{\text{NumberPositiveSamples}} \). The recall indicates how complete the classification is and how many of the actionable alerts will be reported. Like precision, recall alone is not meaningful as a classifier that classifies all samples as actionable would have a perfect recall of 1.

4.4.1 Evaluating the Models
Because the classifiers are designed to match the given data as good as possible, there exists a significant chance of over-fitting the model. If that happens, the model is very good at classifying the given data, but bad at generalizing for new data.

When we evaluate the models for each project separately, we run into this problem as the data used to train the classifiers is the same that is used to test them afterwards. In order to avoid this, we use 10-fold-cross-validation. That means the data is split into ten parts and the model is trained ten times, each time leaving out a different part of the data. This
remaining part is then used to test the model. The average scores with their variance are then recorded to provide accuracy, precision and recall.

This cross-validation is not necessary when we apply a model trained on one project to another one, because the train- and test-sample-sets are not overlapping.
Chapter 5

Evaluation of Features and Classifiers

In this chapter we apply the process of gathering data and extracting features to build and evaluate classifiers as described in Chapter 4. Section 5.1 will introduce the studied software projects. In Section 5.2.1 we will take a look at how the features behave across the chosen projects. This will be followed by the evaluation of the classifiers, first on a single project with 10-fold-cross-validation and then we investigate the cross-project application. As our experiments show that lifetime best predicts actionability, we evaluate a custom classifier based on lifetime alone in Section 5.2.4. We end the chapter with a discussion of our findings.

5.1 Experiment Setup

In the following experiments we follow the model-building process described in Section 4.1 to train 3 different classifiers for three software projects and compare it to a baseline.

We decided to pick small software projects of less than 150k lines of code that were written in the same programming language and shared a common purpose. All these requirements help make the results comparable. Different programming languages have sometimes incomparable bugs, e.g. Java will not be subject to arbitrary pointer arithmetic problems. Additionally, we needed to have access to the source-code repository and thus picked only open-source software projects. In order not to analyze abandoned code we limited the selection further to project that have seen at least a dozen different contributors and were active in the last six months.

We chose three forum-software projects, also known as bulletin-board(BB) software, that met these criteria:
• **FluxBB**[11] is a fork of PunBB, whose first commit in the repository we used was done in 2008 and consists of around 21k lines of code.

• **MyBB**[24] is a more mature project that was initially released in 2002 but after a long period of inactivity development was taken up again in 2010. This is also the earliest commit we analyzed. The latest revision has about 100k lines of code.

• **PunBB**[26] was originally released 2003, but changed developers in 2008, which also marks the day of the earliest commit we analyzed. It is now about 32k lines of code big.

Some other forum software was considered but discarded due to various reasons: PhpBB\(^1\) was roughly 267k lines of code much bigger and infeasible to analyze in a short time. Beehive Forum\(^2\) and Phorum\(^3\) were not considered as they both had a single developer contribute the vast majority of commits. Finally, SimpleMachines Forum\(^4\) was not picked because the development process considers a different branch than master as the main branch, which we did not want to account for as branches already created some problems discussed in Section 3.3.2.

<table>
<thead>
<tr>
<th></th>
<th>FluxBB</th>
<th>MyBB</th>
<th>PunBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>21k</td>
<td>100k</td>
<td>32k</td>
</tr>
<tr>
<td>Actionable</td>
<td>1798</td>
<td>87</td>
<td>1397</td>
</tr>
<tr>
<td>Un-actionable</td>
<td>2207</td>
<td>527</td>
<td>4891</td>
</tr>
<tr>
<td>Deleted</td>
<td>110</td>
<td>0</td>
<td>998</td>
</tr>
<tr>
<td>Total</td>
<td>4115</td>
<td>614</td>
<td>7286</td>
</tr>
</tbody>
</table>

Table 5.1: Unique alerts generated for each project.

In Table 5.1 we see the breakdown of unique alerts generated per project with their respective size. We gathered this data using our Autobugs framework discussed in Chapter 3. While FluxBB and PunBB generated multiple thousand alerts, MyBB, the biggest project, did not. We cannot explain this difference and decided to still include the project to see how it behaves in the evaluation of classifiers.

Whenever we use the same project for training and testing a classifier, we are applying 10-fold-cross-validation as described in Section 4.4.1. The random classifier is the only exception for this approach as it does not require any training data. For all cases where the training- and test-data do not

---

\(^1\)https://www.phpbb.com/
\(^2\)https://www.beehiveforum.co.uk/
\(^3\)http://www.phorum.org/
\(^4\)http://www.simplemachines.org/
come from the same project, we use the complete set of alerts from the source project to train the classifier and the complete set of alerts from the target project to test it.

5.2 Results

5.2.1 Features

Before training and evaluating the classifiers we will visualize the features of all three projects combined to better understand which ones can indicate the actionability of an alert by themselves. In Figure 5.1 we see the histograms of all features, which are described in Section 4.2.

The horizontal axis shows the values of the feature while the vertical plots the number of alerts. The types are plotted on a logarithmic scale in Figure 5.1b.

We see that only the lifetime provides a clear separation of actionable and un-actionable alerts. Some types, as seen in Figure 5.1b are only reporting un-actionable alerts. However this is not as useful as it would be if they were reporting only actionable ones. From the alert not being closed during the lifetime of the project we cannot deduce that the alert should be ignored, because we do not know if the developers were simply unaware of the problem.

The unclear separation of (un-)actionable alerts for all other types is surprising as that neither complexity of a unit nor the severity as reported by Static Analysis Tool A seem to be a direct indicator of actionability. However, the data might perform better by applying more complicated classifiers, as we do in the following section.

5.2.2 Experiment 3A: Train- and Test-Data from Same Project

In this experiment we run the three classifiers described in Section 4.3 on each of the three software projects using 10-fold-cross-validation except for the random classifier. There is no need to cross-validate it because it is not dependent on training data and randomly assigns one of the two possible classes.

Evaluating the Decision Tree

In Table 5.2 we can see the accuracy, precision and recall values for the three projects when using the decision tree. The values are computed as the average of the 10 cross-validation runs and the 95% confidence interval based on the 10 results. We see that the decision tree is very good at predicting the actionability property of alerts based on the features listed in Section
CHAPTER 5. EVALUATION OF FEATURES AND CLASSIFIERS

Figure 5.1: Histograms of the alert-features of all three projects combined.
CHAPTER 5. EVALUATION OF FEATURES AND CLASSIFIERS

<table>
<thead>
<tr>
<th>Decision Tree</th>
<th>FluxBB</th>
<th>MyBB</th>
<th>PunBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0.97 ± 0.08</td>
<td>0.96 ± 0.05</td>
<td>1.00 ± 0.02</td>
</tr>
<tr>
<td>Precision</td>
<td>0.99 ± 0.04</td>
<td>0.89 ± 0.21</td>
<td>1.00 ± 0.00</td>
</tr>
<tr>
<td>Recall</td>
<td>0.93 ± 0.18</td>
<td>0.85 ± 0.41</td>
<td>0.98 ± 0.09</td>
</tr>
</tbody>
</table>

Table 5.2: Scores and 95% confidence interval for the decision tree.

4.2. It is also quite stable with the exception of the precision and recall for MyBB.

Both precision and recall divide the number of true positives, that is the number of alerts labeled actionable which were correctly classified as such, by some other value. In the case of MyBB the number of alerts labeled actionable is much lower. It is very likely that the selection of 10 non-overlapping sets of alerts created testing-sets which do not contain any actionable alerts. in this case the recall is zero, skewing the metric. This artifact of the data is even more visible for the linear support vector machine.

Evaluating the Linear and Non-Linear Support Vector Machine

<table>
<thead>
<tr>
<th>Linear SVM</th>
<th>FluxBB</th>
<th>MyBB</th>
<th>PunBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0.66 ± 0.33</td>
<td>0.86 ± 0.01</td>
<td>0.72 ± 0.46</td>
</tr>
<tr>
<td>Precision</td>
<td>0.68 ± 0.36</td>
<td>0.00 ± 0.00</td>
<td>0.59 ± 0.59</td>
</tr>
<tr>
<td>Recall</td>
<td>0.71 ± 0.58</td>
<td>0.00 ± 0.00</td>
<td>0.86 ± 0.45</td>
</tr>
</tbody>
</table>

Table 5.3: Scores and 95% confidence interval for the linear support vector machine.

As we see in Table 5.3, the linear SVM performs worse than the decision tree classifier (see Table 5.2). The big confidence intervals and the poor performance indicate that the data is not linearly separable and the SVM struggles to find a hyperplane. This is the reason why we also included the non-linear SVM, which performs far better, as visible in Table 5.4. Just as with the decision tree, the classifier has a high variance on the recall and precision for MyBB. Again, this is because the project has very few actionable alerts compared to the un-actionable ones.

<table>
<thead>
<tr>
<th>SVM</th>
<th>FluxBB</th>
<th>MyBB</th>
<th>PunBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0.98 ± 0.01</td>
<td>0.94 ± 0.07</td>
<td>0.93 ± 0.17</td>
</tr>
<tr>
<td>Precision</td>
<td>0.95 ± 0.02</td>
<td>0.90 ± 0.60</td>
<td>1.00 ± 0.00</td>
</tr>
<tr>
<td>Recall</td>
<td>1.00 ± 0.00</td>
<td>0.60 ± 0.53</td>
<td>0.69 ± 0.79</td>
</tr>
</tbody>
</table>

Table 5.4: Scores and 95% confidence interval for the non-linear support vector machine.
Evaluating the Random Classifier

<table>
<thead>
<tr>
<th></th>
<th>FluxBB</th>
<th>MyBB</th>
<th>PunBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0.52</td>
<td>0.52</td>
<td>0.50</td>
</tr>
<tr>
<td>Precision</td>
<td>0.59</td>
<td>0.40</td>
<td>0.42</td>
</tr>
<tr>
<td>Recall</td>
<td>0.49</td>
<td>0.58</td>
<td>0.51</td>
</tr>
</tbody>
</table>

Table 5.5: Scores for the Random classifier.

The performance of the random classifier, as shown in Table 5.5, is as expected poor. It classifies around 50% of the alerts correctly.

5.2.3 Experiment 3B: Cross-Project Performance

In this experiment we extend the prior experiment 3B by training each classifier on a project and using the resulting model to classify all alerts of another project. Figures 5.2c, 5.3c, 5.4c and 5.5c show the resulting scores of these procedures. In case where the training- and test-project is the same we use the 10-fold-cross-validation scores from experiment 3A.

The cross-project values are computed based on all samples of the project. As mentioned in Section 4.4.1 there is no need for 10-fold-cross-validation in these cases.

Evaluating the Decision Tree

In Figure 5.2c we see the resulting scores of training the decision tree across the projects. As discussed in Section 5.2.2, the scores are mostly good per-project but worse when applying a model trained on one project to another one. The reason for this is most likely that the decision tree was build using no limiting parameters, eg. a maximum depth. This causes it to over-fit the training data and badly generalize the classification. It still does better than the random classifier (compare Figure 5.5c).

Evaluating the Linear Support

The results for the linear support vector machine are shown in Figure 5.3c. Suprisingly, the accuracy improves when the model is trained on the FluxBB-dataset and applied to different projects but overall shows poor performance just like in the single-project case of experiment 3A. The LSVM barely outperforms the random classifier and exhibits a complete failure to recall true positives when trained on MyBB. It is very likely that the problem is not linearly seperable. This is why we chose to also include the non-linear
Figure 5.2: Cross-project scores for the decision tree.

(a) Accuracy

(b) Precision

(c) Recall
CHAPTER 5. EVALUATION OF FEATURES AND CLASSIFIERS

Figure 5.3: Cross-project scores for the LSVM.

(a) Accuracy

(b) Precision

(c) Recall
CHAPTER 5. EVALUATION OF FEATURES AND CLASSIFIERS

SVM.

Evaluating the Non-Linear Support Vector Machine

Figure 5.4: Cross-project scores for the SVM.

(a) Accuracy
(b) Precision
(c) Recall

Figure 5.4c shows the resulting scores for the non-linear support vector machine classifier. Although the non-linearity greatly improves the results for the single-project case of experiment 3A, it performs even poorer than the LSVM for cross-project cases. This indicates that the model is overfitting the data and very bad at generalizations and that the non-linearity is not helping the cross-project classification.
CHAPTER 5. EVALUATION OF FEATURES AND CLASSIFIERS  51

Figure 5.5: Cross-project scores for the random classifier.

(a) Accuracy

(b) Precision

(c) Recall
CHAPTER 5. EVALUATION OF FEATURES AND CLASSIFIERS  52

Evaluating the Random Classifier

As expected the scores of the random classifier are around 0.5 and thus consistently worse than the decision tree classifier. However, even a random classification has a higher accuracy than the SVM model trained on the FluxBB data when applied to other project. This shows that the SVM is over-fitting the data and not suitable to be applied across different projects at all.

5.2.4 Experiment 3C: Custom Classifier Based on Lifetime

As we chose the decision tree classifier for its ability to explain its reasoning, we used a visualization of the decision tree to investigate which alert-properties were used to classify the alerts.

For all three project, the first decision the classifier makes is whether the alert is below a certain lifetime. To our surprise, this value is not only very close to three years across all three projects, but also the most deciding factor for classifying an alert. No other property is as important as the lifetime for the decision tree. The complete trees can be found in the Appendix.

Based on this finding, we build a custom classifier that calculates a lifetime which best separates the actionable from the un-actionable alerts. Effectively this is a decision tree that only operates on the lifetime property of an alert and is one node deep. In Table 5.6 we see that is scores similarly to the decision tree (Table 5.2) and Figure 5.6 shows that it outperforms the decision tree classifier for cross-project scores.

<table>
<thead>
<tr>
<th>Custom Classifier</th>
<th>FluxBB</th>
<th>MyBB</th>
<th>PunBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0.96 ± 0.09</td>
<td>0.97 ± 0.05</td>
<td>0.99 ± 0.02</td>
</tr>
<tr>
<td>Precision</td>
<td>0.98 ± 0.02</td>
<td>0.85 ± 0.21</td>
<td>0.98 ± 0.03</td>
</tr>
<tr>
<td>Recall</td>
<td>0.94 ± 0.20</td>
<td>0.97 ± 0.20</td>
<td>0.98 ± 0.09</td>
</tr>
</tbody>
</table>

Table 5.6: Scores and 95% confidence interval for the Custom Classifier.

5.3 Discussion

In these experiments we showed that the decision tree and non-linear SVM machine learning models build from historic alert data provide a good way of classifying alerts based on alert data but suffer from the problem of over-fitting. However, further investigation of the data revealed that the lifetime alone is an almost perfect indicator of actionability for our data. A classifier based on lifetime alone is not only comparable in performance, but even
Figure 5.6: Cross-project scores for the lifetime-based classifier.

(a) Accuracy  
(b) Precision  
(c) Recall
CHAPTER 5. EVALUATION OF FEATURES AND CLASSIFIERS  

better when applied cross-project. For each project, the break-off point for an alert to become un-actionable is when it becomes older than three years old. The reason for this is not clear.

It is possible that after three years the project went into maintenance mode, where only the most severe problems are addressed, but because of the different ages and recent commits it is not very likely.

We suspect that the way of working on these project by volunteer developers influences this value of three years. It might be that developers are involved in these open-source projects for roughly three years before moving on and they focus only on code that they contributed themselves. We did not have time to investigate this hypothesis.

Overall, we suspected the severity and confidence to have a higher correlation to the (un-)actionability than the lifetime as they offer a way to rank the alerts and help developers filter out respectively less important alerts or alerts that are more less likely to be true positives. This was not the case.

As will be discussed in 6, the experiments should be replicated with a larger sample of projects, possibly also studying the domain-specific behavior and controlling factors such as development practices for the projects.
Chapter 6

Future Work

This chapter discusses possibilities that opened with the research conducted. It also summarizes alternative approaches that were recognized but not followed through.

6.1 Alternative Approaches

As mentioned in Section 2.2, there exist a multitude of approaches to improve the usefulness of static analysis tools, from educating developers, improving the user-interface and user-experience of the tools to improving the methods these tools use to find problems. We chose only to investigate post-processing the generated alert lists but feel it is important to mention that there are alternative ways of approaching this problem.

Our initial idea was to include data from the issue-trackers of open source software project to improve the classification of (un-)actionable alerts. However, it proved to be too difficult to find projects that met all our requirements: Around 100k lines of code, written in a language that the static analysis tools we had access to supported and an accessible and scrape-able issue tracker. Our initial intention was to work with the code of major browsers like Firefox or Chrome but they were too big projects to be analyzed. As an interesting side-note, the complete analysis of the Firefox code-base with Static Analysis Tool A would have taken over a millennium.

In this work we only investigated the classification of alerts into action-able and un-actionable ones. This means, that some alerts are filtered out and not presented to developers. Valid problems could be omitted this way and especially if there are few alerts reported to the developers it would be bad to completely hide them. An alternative to classifying is to rank alerts, letting developers decide how many alerts they wish to investigate.
CHAPTER 6. FUTURE WORK

Our ground truth is sourced automatically from historic alert data. We assume that alerts which disappeared were true positives, discovered and intentionally fixed. A qualitative investigation of these assumptions has not been conducted, neither by us nor by similar approaches by Allier et al.\cite{1} or Heckman et al.\cite{14}. It would be worthwhile to evaluate how well these assumptions hold.

The validation of the models has been conducted with the same ground truth. We chose not to conduct developer interviews to evaluate the usefulness of the post-processing due to timing constraints. Similar to evaluating the ground truth, the evaluation of the result of post-processing could be done with more involvement of the people working on the studied software project.

6.2 Using Historic Alert Data

While building and working with our framework Autobugs to collect historic alert data, we found several issues with how alerts are tracked over time. In the multi-branched revision control system like git, we considered only the lifetime on one branch. However, we did not investigate the effects of this decision. See Section 3.3.2 for a more detailed description of the problem. Additionally, both static analysis tools we used implemented an alert-identifier that is based on a hash of various strings. One of those is the containing file-path, which leads refactoring to mark an alert as ‘deleted’ according to our classification. This can have a major impact on the lifetime of an alert.

As mentioned in Section 2.3.3 already, models based on historic alert data could potentially mislead developers to believe they have no problems. When they never fix a certain kind of alert (eg. SQL-injections), the model will classify it as un-actionable further hiding this issue from the developers.

There are many more machine learning models and although it is very modern to use neural networks (NNs) to solve classification problems, we chose not to work with those. It is still very difficult to impossible to understand how a neural network classifies samples and the lack of explanation for the NN is something that we feel is not desirable. Additionally, these models work best when there are tens of thousands of samples, which we did not have. A large-scale repetition of our research with more and bigger projects would be better suited to use NNs.

Investigating these topics would not only benefit our research, but several others using historic alert data.
Chapter 7

Conclusions

In this work we build Autobugs, a framework to collect historic alert data. We used it to replicate parts of an experiment conducted by Heckman et al.[14], by using the FindBugs static analysis tool on the JDOM software project. To investigate how much longer it takes to collect the data from a significantly larger project we used it on Elasticsearch, which is with its 600k lines of code roughly 30 times bigger. Additionally, we included a state of the art commercial static analysis tool to compare the results.

We found that tracking alerts across revisions is non trivial as the methods provided by the tools do not take refactoring into account. Additionally, the possibility to work on multiple branches simultaneously with the version control software git[12] can introduce much uncertainty in the computed lifetime of the alerts. Prior work has not taken this into account and developing more robust methods to track alerts is a venue for future research.

We then proceeded to use Autobugs to gather historic alert data for Static Analysis Tool A on three similar software projects. The alert-data was then combined with metrics on complexity to build a classifier that could predict the actionability of an alert from its properties and the properties of the containing unit. The three classifiers that were evaluated were the decision tree, a linear and a non-linear support vector machine. The results were compared to a random classifier as a baseline. All approaches were evaluated on each project separately as well as the cross-project application of their models.

Although they mostly outperformed the random baseline for separate projects and cross-project application, only the decision tree was exceedingly accurate at classifying alerts as (un-)actionable. Investigating the cause, we found that the best indicator of actionability for all three projects is the lifetime of an alert. However, we found no good explanation for this in our data and suspect it is tied to the development workflow of the projects, which we did not investigate.
CHAPTER 7. CONCLUSIONS

7.1 Discussion

At first the topic of classifying alerts based on historic data seemed very accessible and a lot of opportunities for research were open, but it soon became clear that most of the proposed topics were already covered. It became challenging to find a research question that has not been already answered after doing a literature survey.

Additionally, the surveyed papers were of varying value and required very rigorous reading to find hidden assumptions. Beller et al. [4] for example mentioned the Elasticsearch project[7] using FindBugs[9] as one of three static analysis tools and stated the developers for the project are required to use these tools. It turned out however, that although FindBugs has been used occasionally, it was not mandatory. Another tool was, but the paper did not make a clear distinction between them.

The mandatory use of FindBugs would have provided a reliable source for identifying false positives as every alert generated by the tool would have been registered by the developers. If the use of a tool is not mandatory, there might be quite some time between the revision where a tool generates an alert based on the issue, and the developers becoming aware of the issue, if they even find out about it.

The chosen research approach has multiple unverified assumptions. The first being that alerts which disappear during the lifetime of the project are in fact actionable. It is not researched to which extent this assumption holds. We investigated the approach of correlating alert data to an issue tracking system, but the only projects that had sufficient data available, Firefox[10] and Chrome[6], were too big to be analyzed by Autobugs.

Working with FindBugs we wondered why it was not used for student projects during introductory or advanced programming classes. Once it is set up, it would have provided valuable insight and could greatly improve the produced code as well as give an idea of what flaws one produces while programming. In my opinion the false positive rate would be very low for the small programming exercises usually required for these classes. It is surprising how many alerts the tools generated for very actively developed software projects.

The plotting of alert-lifetimes was very helpful in identifying problems with our approach. It helped to recognize bugs very early on in the development of Autobugs and showed the artifacts discussed in experiments 1 and 2. It would have been much more difficult to work with the data if it had not been visualized.
Bibliography


[29] The Linux kernel Archives. URL: https://www.kernel.org/ (visited on 04/03/2017).
Appendix

Most Recent Revisions of the Analyze Software Projects

<table>
<thead>
<tr>
<th>Project</th>
<th>Revision Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>jdom</td>
<td>2c4bf202c21bd25606b6c9ad6e13795a1e8166d1</td>
</tr>
<tr>
<td>elasticsearch</td>
<td>c1e5421b7718a918885e7e5f22ab4e0b7bbbd6fc6</td>
</tr>
<tr>
<td>fluxbb</td>
<td>5a9a5f401f1711ecb610d5e6a70bf8c0d1e3847a</td>
</tr>
<tr>
<td>mybb</td>
<td>883effc24774ca05ea94b9fc258ac34ec19a6eae</td>
</tr>
<tr>
<td>punbb</td>
<td>b20a80ae87ec5285df5d0e0de1bb64e5923e603f</td>
</tr>
</tbody>
</table>

Generated Decision Trees
Figure 1: Decision tree trained on FluxBB alert-data
Figure 2: Decision tree trained on MyBB alert-data
Figure 3: Decision tree trained on PunBB alert-data