

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Software Repository Mining Analytics to Estimate Software Component Reliability

André Freitas - freitas.andre@fe.up.pt

 U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rui Maranhão - rma@fe.up.pt

Co-Supervisor: Alexandre Perez - alexandre.perez@fe.up.pt

July 26, 2015

Software Repository Mining Analytics to Estimate Software Component Reliability

André Freitas - freitas.andre@fe.up.pt

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Doctor Hugo José Sereno Lopes Ferreira

External Examiner: Doctor Jâcome Miguel Costa da Cunha

Supervisor: Doctor Rui Filipe Maranhão de Abreu

July 26, 2015

Abstract

Finding and fixing software bugs is expensive and has a significant impact in Software development effort. Repositories have hidden predictive information about Software history that can be explored using analytics and machine learning techniques. A Software component can be a file, class or method in terms of granularity. Current research in Mining Software Repositories (MSR) is capable of ranking and listing faulty components at the file granularity. Crowbar is an automatic Software debugging tool that uses a technique named Barinel. Our goals are predicting Software defects with method granularity and improve Crowbar, by mining repositories.

We have implemented a tool named Schwa, available for free on Github, that is capable of analyzing Git repositories. We are analyzing metrics such as revisions, fixes, authors and the time of commits to feed the prediction model. The analysis of time provides a method to ignore old components. Experimental results shown that for every Software repository, the predictive power of each metric is different. For example, in some projects revisions is more correlated with future defects and in others is fixes. The usage of defect predictions from Schwa in Crowbar reduced the amount of time necessary to rank faulty components. In the Joda Time project the time was reduced from one hour to less than a minute.

This thesis does the following contributions: a method to parse and represent diffs from patches with method granularity for Java; a model to compute defect probabilities; a framework for mining Software repositories; a technique to learn the importance of tracked metrics; a method to evaluate the gain of using defect probabilities in fault localization.

Resumo

Encontrar e corrigir bugs tem um grande custo e impacto no esforço em desenvolver Software. Os repositórios escondem informação preditiva sobre o histórico de Software que pode ser explorada recorrendo a técnicas de análise e de machine learning. Um componente de Software pode ser um ficheiro, classe ou método em termos de granularidade. A investigação atual de Mining Software Repositories (MSR) é capaz de classificar e listar componentes defeituosos com a granularidade ao nível do ficheiro. O Crowbar é uma ferramenta que faz depuração automática de Software e usa a técnica Barinel. Os nossos objetivos são prever defeitos em Software com granularidade até ao método e melhorar o Crowbar, ao extrair informação de repositórios.

Foi implementada uma ferramenta denominada de Schwa, disponível livremente no Github, que é capaz de analisar repositórios Git. Estamos a analisar métricas como as revisões, correções de bug, autores e o tempo dos commits para alimentar o modelo de previsão. A análise do tempo permite ignorar componentes mais antigos. Os resultados experimentais demonstraram que para cada repositório de Software, o poder preditivo de cada métrica é diferente. Por exemplo, em alguns projetos o número de revisões está mais correlacionado com futuros defeitos e em outros é o número de correções de bugs. A utilização das previsões de defeito do Schwa no Crowbar reduziu o tempo necessário para classificar componentes faltosos. No projecto Joda Time o tempo foi reduzido de uma hora para menos de um minuto.

Esta tese faz as seguintes contribuições: um método para interpretar e representar diffs de patches com a granularidade ao método; um modelo para calcular probabilidades de defeito; uma framework para minar repositórios de Software; uma técnica para aprender a importância das métricas analisadas; um método para avaliar o ganho de usar as probabilidade de defeito em localização de falhas.

Acknowledgements

First, I would like to thank my supervisor and co-supervisor, Rui Maranhão and Alexandre Perez for their extraordinary help and mentoring in my dissertation, specially for supporting me on the most difficult challenges. Thanks for accepting me as a dissertation student and for your patience through the last months. I would like to thank my supervisor for the financial aid I received through FCT funding, since it was an important help for me. I would like to thank Nuno Cardoso for helping me through the internals of Crowbar.

Regarding my experiments, I would like to thank the contributions from Shiftforward, Luís Fonseca, Diogo Pinela and Strongstep. Thanks Open Source community for making available software for free, that students and researchers frequently use on their projects. Thanks FEUP for having a good environment, teachers and for everyone that indirectly contributed to the success of this thesis that I could not list here.

Finally and not least important, I would like to thank my parents, my family and my girlfriend for always supporting me. They surely gave me an environment to be a better person and pursuing my goals.

André Freitas

“Imagination is more important than knowledge. For knowledge is limited to all we now know and understand, while imagination embraces the entire world, and all there ever will be to know and understand.”

Albert Einstein

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation and goals	2
1.3	Concepts and definitions	2
1.3.1	Software testing glossary	2
1.3.2	Types of tests	2
1.3.3	Defect prediction	3
1.3.4	Fault localization	3
1.4	Problem statement	3
1.5	Contributions	3
1.6	Document overview	4
2	State of the art	5
2.1	Best practices and recommendations	5
2.1.1	Data extraction and preparation	5
2.1.2	Synthesis	5
2.1.3	Analysis	6
2.1.4	Sharing results	6
2.2	Research on Mining Software Repositories	6
2.2.1	History slicing	6
2.2.2	Identifying types of bugs	6
2.2.3	Whosefault	7
2.2.4	Classification of bugs	7
2.2.5	Mining Git repositories tools	7
2.3	Defect prediction techniques	8
2.3.1	Approaches	8
2.3.2	Metrics	8
2.3.3	Time-Weighted Risk	9
2.3.4	Fixcache and Bugcache	11
2.3.5	Change Classification	12
2.4	Fault localization techniques	12
2.4.1	Program-spectra based	12
2.4.2	Model-based	13
2.4.3	Barinel	14
2.4.4	Crowbar	15
2.5	Similar tools	16
2.5.1	Codacy	17
2.5.2	Moskito	17

CONTENTS

2.5.3	SensioLabsInsight	18
2.5.4	Code Climate	19
2.5.5	Pull Review	20
3	A Technique to Estimate Defect Probabilities	21
3.1	Schwa	21
3.1.1	Installation	21
3.1.2	Usage	21
3.1.3	Visualization	22
3.1.4	Process	23
3.2	Features weight estimation	27
3.3	Diagnostic cost	28
3.3.1	Example	28
3.3.2	Schwa integration with Crowbar	29
4	Experimental results	31
4.1	Features weight estimation	31
4.1.1	Experimental setup	31
4.1.2	Results	32
4.2	Diagnostic cost	36
4.2.1	Experimental setup	37
4.2.2	Results	37
5	Discussion	43
5.1	Features weight estimation	43
5.2	Diagnostic cost	44
5.3	Threats to validity	44
6	Conclusions and Further Work	45
6.1	Goals satisfaction	45
6.2	Further work	45
	References	47

List of Figures

2.1	Time Weighted Risk (TWR)	10
2.2	Hit-spectra matrix example	14
2.3	Crowbar sunburst chart report	16
2.4	Codacy dashboard	17
2.5	Moskito dashboard	18
2.6	Example of a report from SensioLabsInsight	19
2.7	Issues listed on Code Climate	20
2.8	Pull Review report	20
3.1	Sunburst results for Joda Time	22
3.2	Process pipeline	23
3.3	Repository Model	24
3.4	Modified Time Weighted Risk (TWR)	26

LIST OF FIGURES

List of Tables

2.1	Repository example - files and commits	10
2.2	Repository example - commits	10
2.3	Program spectrum	13
3.1	Examples of ranks	28
4.1	Schwa results	32
4.2	Libcrowbar results	32
4.3	Joda Time results	33
4.4	Meo Arena results	33
4.5	Mongo Java driver results	33
4.6	Scraim results	34
4.7	Trainsim results	34
4.8	Adstax results	34
4.9	Boxer results	35
4.10	Apso results	35
4.11	Hivedb results	35
4.12	Teamengine results	35
4.13	Automatalib results	36
4.14	CDI TCK results	36
4.15	Commits applied to Joda Time	37
4.16	Schwa configurations for Joda Time	38
4.17	Diagnostic cost for Joda Time	39
4.18	Commits applied to CDI TCK	39
4.19	Schwa configurations for CDI TCK	40
4.20	Diagnostic cost for CDI TCK	40

LIST OF TABLES

Abbreviations

MSR	Mining Software Repositories
TWR	Time-Weighted Risk
SCM	Source Control Management
SFL	Spectrum-based Fault Localization
MDB	Model Based Diagnostic
ISTQB	International Software Testing Qualifications Board
SaaS	Software as a Service
LRU	Least Recented Used
URL	Uniform Resource Locator
PHP	PHP: Hypertext Preprocessor
CSS	Cascading Style Sheets
HTML	HyperText Markup Language
MIT	Massachusetts Institute of Technology

Chapter 1

Introduction

We review the state of the art in Mining Software Repositories (MSR), existing tools and propose a new method to predict defects based on data extracted from repositories. Also we use this information from defect prediction to improve the diagnostic accuracy from Crowbar, namely, the Barinel algorithm.

1.1 Context

Software plays an important role for society and in our daily routine, since we use applications to communicate, manage information, etc. We expect that these applications behave correctly and we are easily frustrated when they are defective. Development of software is not a simple task since developers need to maintain complex code, test and manage expectations of stakeholders by correctly interpreting requirements. It is estimated that fixing bugs represent 90% of development costs [Ser13].

There are tools that can help developers delivering high quality software, by automatically reviewing code and analyzing their behaviour. Some of these tools are Codacy¹, Crowbar² and Codeclimate³. The usage of revision control systems such as Git, SVN and Mercurial, helps developers tracking changes on Software and understanding the evolution of components. Tools are important to developers since they automate and avoid repetitive tasks in Software development.

With the growing usage of revision control systems, research in MSR evolved in the last decade and involves the analysis of systems used to support the development of software such as repositories, issue trackers and mailing lists [HNB⁺13].

¹<https://codacy.com/>

²<http://crowbar.io/>

³<https://codeclimate.com/>

1.2 Motivation and goals

Software repositories have hidden information that can be explored with analytics and machine learning techniques to support defect prediction models. The Barinel algorithm in Crowbar uses static estimations for defect prediction and goodness of components [AZG09]. Insights from software history could substitute these estimations and improve the diagnostic accuracy with more dynamic estimations.

Our goals are:

- Predict defects from Software repositories by learning what are the most important features to analyze and create a prediction model based from existing techniques;
- Improve diagnostic accuracy of Barinel with defect prediction probabilities.

1.3 Concepts and definitions

Software quality is the degree to which the system meets requirements and expectations of users. The main characteristics of product quality, by ISO/IEC 25010, are portability, maintainability, security, reliability, functional stability, performance efficiency, compatibility and usability [ISO11].

1.3.1 Software testing glossary

According to ISTQB ⁴, the standard glossary used in software testing is:

Error

A human action that produces an incorrect result;

Fault, defect, bug

Flaw in a component that causes the system to fail performing its required functions;

Failure

Deviation of the component from its expected result.

1.3.2 Types of tests

Testing is the process, consisting of static and dynamic activities, that involves the planing, preparation and evaluation of software to check if it meets the requirements and to detect defects.

Static testing

Is the analysis of the system static representation such as source code and documents, improving the internal quality.

⁴<http://istqb.org>

Dynamic testing

It involves the execution of the system, observing its behavior under test cases, improving external quality.

1.3.3 Defect prediction

Defect prediction consists in reliably predicting software defects using information from code metrics, process metrics or previous defects [DLR12]. Some approaches use binary classification, asserting if a component is defective or not, while others rank them by giving a score.

1.3.4 Fault localization

Fault localization consists in finding the component that caused the Software to fail. Is one of the most time consuming activities in debugging. Considering this, there is a high demand for making this process automatic, leading to the development of techniques that makes this activity more effective [WD09].

1.4 Problem statement

Considering our goals, we want to answer the following research questions:

RQ1

What features should we extract and analyze from Software Repositories to predict defects?

RQ2

Can we improve Barinel fault localization technique from Crowbar with the results from defect prediction?

1.5 Contributions

This thesis makes the following contributions:

- A technique to parse and represent diffs from patches achieving method granularity in Java;
- A model to compute defect probabilities;
- A framework for mining Software repositories and reporting analytics in a graphical visualization;
- A technique to learn the importance of tracked metrics;
- A method to evaluate the gain of using defect probabilities in fault localization, namely the Barinel algorithm in Crowbar.

1.6 Document overview

1. Introduction

An introduction to the context, goals and concepts of this thesis.

2. State of the art

Current state of the art techniques on Mining Software Repositories and Software Debugging are reviewed, along with example of existing tools.

3. A Technique to Estimate Defect Probabilities

It is presented the tool created to conduct this research and the methodologies used.

4. Experimental results

The experimental setup and results are presented in this chapter.

5. Discussion

A chapter dedicated to the discussion of findings about the results. Initial research questions are answered.

6. Conclusions and Further Work

The conclusions and satisfaction of goals are discussed, along with an overview of further work.

Chapter 2

State of the art

In this chapter it is described the current state of the art in Mining Software Repositories (MSR), by reviewing findings and discussing existing defect prediction techniques. Related tools are presented and Crowbar, the fault localization tool that we aimed to improve with the defect prediction results.

2.1 Best practices and recommendations

Hemmati *et al.* analyzed the last decade of mining software repositories publications, between 2004 and 2012, and produced an article with best practices and recommendations to new researchers in the MSR community [HNB⁺13]. The main activities of Mining Software Repositories research are: data extraction and preparation, synthesis, analysis and sharing results.

2.1.1 Data extraction and preparation

In data extraction the source code is the most important artifact, although communication artifacts can be used such as emails and issue trackers like Bugzilla. The problem in this phase is the use of wrong assumptions since it is important to understand how the Control Version System is used, the project and its domain since exists noisy data [HJZ12]. For example, a commit message may not reflect the change and can be used only as a way of communication. Also, the study of developers and their behavior can produce valuable information, but the problem of multiple online personas representing the same person should be considered.

2.1.2 Synthesis

Synthesis is the phase that involves the prediction and machine learning algorithms that are feed from the extracted data. If we are doing regression analysis, that is estimating the relationship of variables, it is important to be aware of the assumptions used.

2.1.3 Analysis

Results of the synthesis phase are analyzed and interpreted, being considered the most important part of MSR research. A manual inspection of the analysis outputs is required because the use of heuristics and automation may be inaccurate. Since may not be feasible analyzing everything, developers can use just sample data. If prediction or classifiers are used in the synthesis phase, the effectiveness is enough to evaluate the results (recall and recognition).

2.1.4 Sharing results

In MSR sharing results is usually ignored because most of research is based on empirical studies and do not share data. It is suggested that the raw and processed data should be shared along with the tools used, to push the community forward. In 2005 the number of published papers in MSR was 9 and in 2012 was 44 [HNB⁺13]. The themes with most publications are data extraction and modeling.

2.2 Research on Mining Software Repositories

There is relevant work in MSR research capable of extracting insights about repositories, such as finding who is the best developer to fix a bug [Ser13]. This research is also published in the MSRconf¹, that is an annual conference that joins researchers in this area of study.

2.2.1 History slicing

Servant and Jones developed a technique called history slicing that enables developers to track code evolution at the line of code level, since current SCMs require a considerable developer effort to analyze code changes at this granularity [SJ12]. This technique provides the minimal amount of information about the code changes and its implementation is called CHRONOS. The motivation behind this technique is that, by observation, researchers concluded that developers ask often about the last code changes and the complete history to understand why a software component was implemented in a certain way.

2.2.2 Identifying types of bugs

Chadd C. Williams and Jeffrey K. Hollingsworth proposed a technique that compares code changes and the types of bugs being fixed [WH05]. This approach does not analyze commit messages because it is hard to correlate bug reports and code changes. The most common types of bugs are analyzed, such as functions that do not check if a variable is null, and then the list of warnings is ranked to avoid false positives. Further work is necessary to improve this technique such as analyzing other types of bugs.

¹<http://msrconf.org>

2.2.3 Whosefault

Francisco Servant proposed an approach that helps to find the most suitable developer to fix a bug and where the bugs are located [Ser13]. The algorithm to find what is the right developer to fix the bug is called Whosefault and depends on statistical coverage-based fault-localization techniques. This algorithm have an accuracy of up to 37%.

2.2.4 Classification of bugs

Classification of bugs is an important problem in MSR due the presence of noise. For example, a commit can be described as a fix on the message but can be also introducing new features. To avoid mixing new features and bug fixes in the same patch, some standards can be enforced to ensure they are classified the right way on the issue tracker ².

The estimated impact of misclassification in defect prediction, is flagging components as defective that do not have any bugs, on an average of 39% [HJZ12]. The quality of diagnosing defective components depends on the quality of data. Kim Herzig *et al.* found data quality issues, after investigating the projects HTTPClient, Jackrabbit, Lucene-Java, Rhino and Tomcat5 [HJZ12]:

Issue reports classifications are unreliable

In the issue trackers investigated, at least 40% of reports are misclassified.

Every third bug is not a bug

33,8% of bug reports are not bugs.

These researchers also pointed that the main source of wrong reports are related to the fact that developers and users have different perspectives on bug classification.

2.2.5 Mining Git repositories tools

Mining a repository is the first step to extract data to feed defect prediction models. Git repositories usage is growing and are different from SVN repositories because they are decentralized. The advantage of this characteristic is that the information extraction is done locally [SLL⁺11]. Some tools were developed to extract information from Git but the problem is that they are poorly documented, project specific and there are no standards about how to share mining results [Car13].

There are some existing tools that extract data from Git repositories:

git_mining_tools

It is developed in Ruby, extracts data from a Git repository and exports to a PostgreSQL or MySQL database. The source code is available on Github ³.

²<https://docs.python.org/devguide/patch.html>

³https://github.com/cabird/git_mining_tools/

gitdm

It is developed in Python and receives the output from the `git log` command and then generates an HTML document with a report. It gathers statistics from Linux Kernel patches. The repository of this tool is public ⁴.

2.3 Defect prediction techniques

The estimation of component's reliability helps us evaluate from a set of components what are the most bug-prone and then focusing the resources to fix them ⁵. Despite the existing research, organizations still ask how they can evaluate the quality of their software [FN99].

It is important to note that developers must consider that defect prediction is not completely accurate and it should be combined with other practices.

2.3.1 Approaches

Defect prediction approaches can be divided into three main categories [DLR12]:

Change log

It uses metrics from control version systems and use assumptions, such as: the files with most changes are must bug-prone;

Single-version

It analyzes the current state and behavior of the system;

Effort-aware

This approach does not predict if components are buggy, but estimates the ones that require more effort to inspect bugs.

The approaches that are most interesting to us, according to our goals of mining software repositories, are change log approaches. There is a mindmap, that gives an overview of the different defect prediction approaches that is published in Mindmeister⁶.

2.3.2 Metrics

There are metrics to analyze such as Process, Previous defects, Source code and Entropy of Changes [DLR12].

Process

Bugs are caused by changes and for each file the metrics can be the number of revisions, fixes, authors, refactorings, etc [MPS08]. The number of revisions and fixes are the ones that perform better [ZPZ07, GKMS00].

⁴[git://git.lwn.net/gitdm.git](https://git.lwn.net/gitdm.git)

⁵<http://www.cse.ust.hk/~hunkim/Research.html>

⁶<https://www.mindmeister.com/506039387>

Previous defects

Past defects predict future defects. The metrics can be the number of past fixes and the categories of bugs, according to a severity scale. Zimmermann et al. concluded that there exists a high correlation between previous and future defects [ZPZ07].

Source code

Complexity of components is correlated with the effort of changing them. The metrics used are lines of code, object oriented metrics (e.g. number of attributes) and Chidamber & Kemerer metrics (e.g. Lack of cohesion in methods) [CK94].

Entropy of changes

Complex changes are usually more bug-prone. For example, a commit that changes only one file is simpler than one that changes multiple files.

A study concluded that for flagging components that are buggy (binary classification), process metrics perform better but, for ranking components, source code metrics are better suited [DLR12].

2.3.3 Time-Weighted Risk

A technique used at Google is Time-Weighted-Risk (TWR). It is a simple way of estimating component's reliability, easy to understand and a result of a case study [LLS⁺13]. In this model, components are files and those files have commits in certain timestamps. Therefore, for each component a score is computed and those that have the highest score are less reliable so they might have more bugs. The formula is the following:

$$Score = \sum_{i=0}^n \frac{1}{1 + e^{-12ti+12}} \quad (2.1)$$

The score is a sum of all the bug-fixing commits given their normalized timestamps ti , from 0 to 1 where 0 is the start of the repository and 1 it is the last committed time. This model gives more importance to the most recent changes because it uses time to compute the score, so it is able to distinguish components with the same number of fixes if they were fixed in different timestamps. The curve of TWR function 2.2 is described in figure 2.1.

$$twr(t_i) = \frac{1}{1 + e^{-12t_i+12}} \quad (2.2)$$

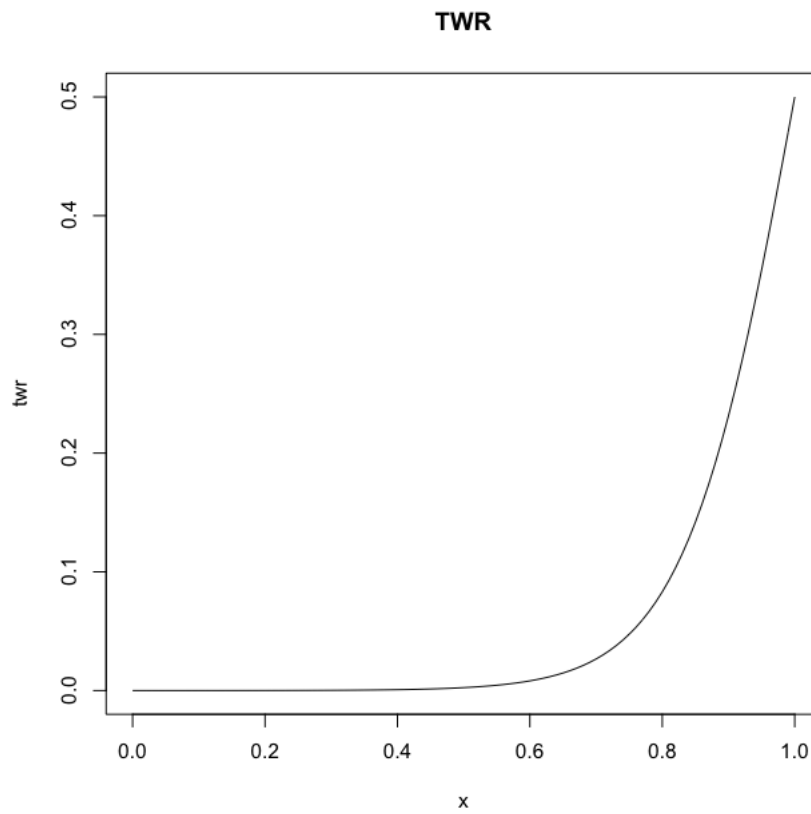


Figure 2.1: Time Weighted Risk (TWR)

2.3.3.1 Example

Given the repository represented in tables 2.1 and 2.2:

File	Commits
main.java	A,B,C
gui.java	A,D
core.java	A

Table 2.1: Repository example - files and commits

Commit	Time	Bug fix?
A	5	no
B	10	no
C	15	yes
D	30	yes

Table 2.2: Repository example - commits

Considering the current timestamp is 50 and the one of the beginning of the repository is 5, the normalized timestamp for the bug-fixing commits are the following:

$$t_{iC} = \frac{15 - 5}{50 - 5} = \frac{2}{9} \quad t_{iD} = \frac{30 - 5}{50 - 5} = \frac{5}{9} \quad (2.3)$$

Using the previous timestamps, the scores for each file are:

$$Score_{main} \approx 0.000088 \quad Score_{gui} \approx 0.004805 \quad Score_{core} = 0 \quad (2.4)$$

Considering this model, the less reliable component is `gui.java` because it has the highest score due the fact that exists a recent bug-fixing commit. If someone asked you to find bugs in a repository you would do the same: inspecting the files with the most commits related to bugs.

2.3.3.2 Discussion

TWR gives more importance to the most recent changes and that is a way of not flagging old untouched files that developers do not want to fix. But, the model is not complete enough because files that have score equal to zero may have unfixed bugs. Therefore, these model should be improved with information from others techniques.

Research has been done at Google where developers had the opportunity to choose between this model and Fixcache (further discussed) and by seeing the results of both of the algorithms, developers chose TWR, mainly because it considers the most recent changes [LLS⁺13]. The study revealed developers are afraid of old source code so a model that highlights those old components cannot be considered good.

2.3.4 Fixcache and Bugcache

Fixcache and Bugcache are defect prediction models based on cache and the difference from other models is that they are more dynamic and have more predictive power [KZWJZ07]. This models assume that faults do not occur in isolation but in burst with other related faults.

A cache is a list of components where the granularity can be the executable binary, module, file or method (entity). Since faults appear in bursts, the cache is updated by the following policies:

- new or modified files may have bugs (new and changed locality);
- buggy files can contain more bugs (temporal locality);
- files changed with buggy files may have bugs (spacial locality).

The size of the cache is fixed and when it is full, a policy of removal should be chosen: the component to remove can be the least recent used (LRU), number of recent changes, number of recent bugs and the number of authors. LRU is the policy that performs better [SLL⁺11].

Fixcache updates the cache when the bug is fixed and Bugcache when the bug is introduced. The bug-fixing changes are detected by mining the repository commits and bugs database. Bug-introducing changes are detected by the bug-fixing changes. For example, if file A has been fixed in a certain commit, the commit that created this file is a bug-introducing change. Due the presence of false positives, this technique can be improved by using bug databases.

A study concluded that Fixcache not only can do bug prediction in a certain point of time but can predict with accuracy at weekly intervals [SLL⁺11].

2.3.5 Change Classification

Change Classification model evaluates if a change will introduce a bug using machine learning techniques by learning from previous bugs [KWZ08]. A problem that may arise is the effect of noise in the training set, compromising the recall and recognition of buggy changes, but techniques are available to reduce this noise. The classifier decides if a change is buggy or clean with 78% accuracy and 60% percent bug recall on average [KWZ08]. This technique has less prediction power but knowing that a commit introduced a bug it is still useful. The main characteristics are:

- classify changes at the file level as buggy or clean;
- detect when a bug is introduced and not when it is fixed;
- takes advantage of source code information (features);
- independent of the programming language using bag-of-words methods.

Support Vector Machines is the approach used to classify changes due its performance in text classification applications. Change Classification can be used as a commit checker, a bug indicator on source code editing and can change the software engineering process by giving immediate feedback to trigger a code inspection when a commit is made [KWZ08].

Some limitations are important to consider such as the process of extracting features that depends how developers used Git and like other machine learning approaches, it takes time to learn.

2.4 Fault localization techniques

Fault localization is the process of finding the component causing the software execution deviating from its expected result. Traditionally, developers use manual techniques just as injecting prints to debug values or breakpoints. Automatic fault localization techniques are used to reduce the cost of finding these faulty components.

2.4.1 Program-spectra based

Program-spectra based methods evaluates the probability of each component being faulty by analyzing the program execution history [PAW14]. It is a statistical technique that, for each test case,

computes the spectrum, that is the code coverage and execution result. The program spectrum is then the list of test cases execution results and can be easily understood by the example in table 2.3.

Test Case	Component A	Component B	Component C	Result
T1	X			Success
T2	X	X		Failure
T3		X		Failure
T4	X		X	Success

Table 2.3: Program spectrum

To determine what components are called in each test case it is used code instrumentation. The program spectrum in table 2.3 use binary flags so it is called hit spectra. From this input, the components that most affects program execution are computed, calculating similarity coefficients using the Ochiai formula [AZG09]. This technique also exploits information from execution outcomes. The output is then the similarity coefficient for each component, that is their likelihood of containing the fault.

Spectrum Fault Localization (SFL) approaches have a good quality of diagnosis and scale well, but is more accurate when the system have many test cases [MS08]. There are some tools based on SFL such as:

Gzoltar

An eclipse plugin⁷ that integrates with JUnit tests [CRPA12].

Tarantula

A technique and a tool⁸ that offers recommendations to reduce the time needed to find the fault [JH05].

2.4.2 Model-based

Model-based techniques use reasoning to do fault localization by having the system knowledge a priori. The system model, the description of correct behavior, is used to compare with the observed behavior of the program and then the difference is used to identify components that explain this deviation [MS08]. Since model-based approaches in software engineering require a formal specification, to avoid this limitation the model is obtained by inference from test cases [PAW14]. The type of model can be:

- based on dependencies between program statements;
- based on computing values propagation;

⁷<http://gzoltar.com/>

⁸<http://spideruci.org/fault-localization/>

- based on creating abstraction models for particular fault assumptions.

Model-based fault localization should be combined with others techniques since the current approaches are not efficient, with high computational cost and scale poorly.

2.4.3 Barinel

Barinel is a combination of Spectrum-based fault localization and Model Based Diagnostic [AZG09]. It starts by receiving a hit-spectra matrix, that contains the observation of running the test cases.

	obs			e
	c ₁	c ₂	c ₃	
t ₁	1	1	0	1
t ₂	0	1	1	1
t ₃	1	0	0	1
t ₄	1	0	1	0

Figure 2.2: Hit-spectra matrix example

Figure 2.2 shows an example of a hit-spectra matrix, with the outcome e of every test case t and the components involved. For example, test case t_1 hits components $\{c_1, c_2\}$ and fails.

The algorithm then takes the following steps:

Candidate generation

Only minimal candidates are generated. A candidate d is a set of components that explains the observed behaviour of the program. In this example, the list of candidates are:

- $d_1 = \{c_1, c_2\}$
- $d_2 = \{c_1, c_3\}$

Candidate ranking

Each candidate d is evaluated by computing the posterior probability using the Naïve Bayes rule:

$$\Pr(d \mid obs, e) = \Pr(d) \cdot \prod_i \frac{\Pr(obs_i, e_i \mid d)}{\Pr(obs_i)} \quad (2.5)$$

The denominator $\Pr(obs_i)$ is a term that is normalized for all candidates and it is not used for ranking. Let p_j denote the prior probability of a component being faulty. Then, the prior $\Pr(d)$ of a candidate d is:

$$\Pr(d) = \prod_{j \in d} p_j \cdot \prod_{j \notin d} (1 - p_j) \quad (2.6)$$

Let g_j denote the probability of a component behaving normally (goodness). Then $\Pr(obs_i, e_i | d)$ is computed by:

$$\Pr(obs_i, e_i | d) = \begin{cases} \prod_{j \in (d \cap obs_i)} g_j & \text{if } e_i = 0 \\ 1 - \prod_{j \in (d \cap obs_i)} g_j & \text{otherwise} \end{cases} \quad (2.7)$$

If for a certain component g_j is not available, it is computed by maximizing $\Pr(obs, e | d)$ (Maximum Likelihood Estimation (MLE)), for the Naïve Bayes classifier. Considering our example, the probabilities for both candidates d_1 and d_2 are:

$$\Pr(d_1 | obs, e) = \overbrace{\left(\frac{1}{1000} \cdot \frac{1}{1000} \cdot \left(1 - \frac{1}{1000} \right) \right)}^{\Pr(d)} \times \overbrace{\underbrace{(1 - g_1 \cdot g_2)}_{t_1} \times \underbrace{(1 - g_2)}_{t_2} \times \underbrace{(1 - g_1)}_{t_3} \times \underbrace{g_1}_{t_4}}^{\Pr(obs, e | d)} \quad (2.8)$$

$$\Pr(d_2 | obs, e) = \overbrace{\left(\frac{1}{1000} \cdot \frac{1}{1000} \cdot \left(1 - \frac{1}{1000} \right) \right)}^{\Pr(d)} \times \overbrace{\underbrace{(1 - g_1)}_{t_1} \times \underbrace{(1 - g_3)}_{t_2} \times \underbrace{(1 - g_1)}_{t_3} \times \underbrace{g_1 \cdot g_3}_{t_4}}^{\Pr(obs, e | d)} \quad (2.9)$$

By performing MLE for both functions:

- $\Pr(d_1 | obs, e)$ is maximized for $g_1 = 0.47$ and $g_2 = 0.19$;
- $\Pr(d_2 | obs, e)$ is maximized for $g_1 = 0.41$ and $g_3 = 0.50$.

Applying the computed values for goodness, $\Pr(d_1 | obs, e) = 1.9 \times 10^{-9}$ and $\Pr(d_2 | obs, e) = 4.0 \times 10^{-10}$. The ranking is then (d_1, d_2) .

2.4.4 Crowbar

Crowbar⁹, formerly known as Gzoltar, is a tool for Java projects that relies on test cases (dynamic analysis) to help developers locate where is the fault of a bug. It uses the Barinel algorithm, combining Spectrum-Based Fault Localization and Model-Based approaches. It supports granularity until the statement level and use code instrumentation by injecting probes in the source code.

⁹<https://crowbar.io>

State of the art

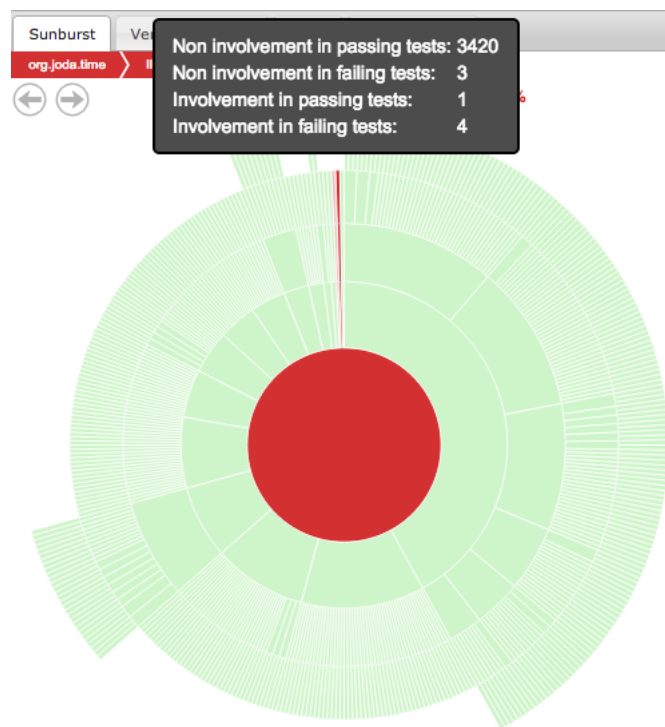


Figure 2.3: Crowbar sunburst chart report

Figure 2.3 shows an example of a sunburst report that can be zoomed. In this type of visualization, the granularity of components increases from the center to the exterior (statement) and the colors are a clue to find faulty components. We also have the possibility to visualize the report as a vertical partition.

Usage

Crowbar supports Junit¹⁰ and TestNG¹¹. It runs as a Java agent on the test suite through the Maven¹² Surefire Plugin¹³. Configuration is done in the file pom.xml¹⁴, that contains information for Maven about the project and configuration details to build and test.

Crowbar will then display the results on a web server by displaying an URL that we must access on a browser to be able to see the report.

2.5 Similar tools

There are plenty of tools available that analyse Software projects. They evaluate code quality and warn developers about problems detected.

¹⁰<http://junit.org/>

¹¹<http://testng.org/>

¹²<https://maven.apache.org/>

¹³<https://maven.apache.org/surefire/maven-surefire-plugin/>

¹⁴<https://maven.apache.org/guides/introduction/introduction-to-the-pom.html>

2.5.1 Codacy

Codacy ¹⁵ is an automatic software revision service (static analysis) that uses defect prediction models to estimate software component reliability. It is a Startup based in Lisbon that won the best pitch award in 2014 Web Summit in London and it is available with free (Open Source) and paid plans.

This service uses the Change Classification principle, classifying a commit as buggy or clean. It supports Scala, Javascript, Python, PHP and CSS. With a few steps it is possible connecting our repository, hosted at Bitbucket or Github.

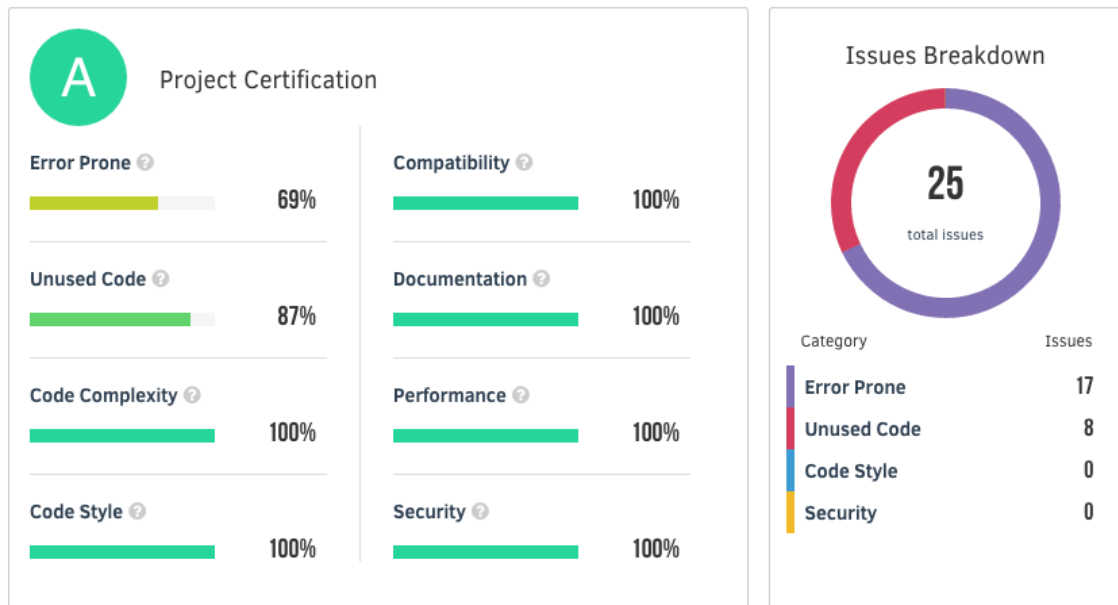


Figure 2.4: Codacy dashboard

Figure 2.4 shows the dashboard with a variety of metrics, reporting a score for code style, errors, code complexity, performance, unused code, compatibility, etc. New and fixed issues are presented, therefore, developers feel rewarded and motivated to improve code quality, a feature that somehow failed in the tool developed in a research conducted at Google in 2013 by Chris Lewis et al. [LLS⁺13].

2.5.2 Moskito

Moskito¹⁶ is a tool that monitors Java Web applications, does fault localization and it is free and Open Source¹⁷. Developers must use annotations to declare what classes or methods to monitor and it does not require changing code, which make this solution simple to use. Two main elements

¹⁵<https://codacy.com>

¹⁶<http://moskito.org>

¹⁷<https://github.com/anotheria/moskito-control>

State of the art

of this tool are the agent and server, where the first collects data and sends it to the server, that processes and displays information in a dashboard.

```
1 //simply add @Monitor
2 @Monitor
3 public class MonitoredClass {
4     public void firstMethod(){
5         //do something
6     }
7     public void secondMethod(){
8         //do something else
9     }
10    //you can also exclude methods from monitoring:
11    @DontMonitor
12    public void doNotMonitorMe () {
13    }
14 }
```

Listing 2.1: Usage example from Moskito documentation



Figure 2.5: Moskito dashboard

The dashboard at figure 2.5 displays performance charts taken from multiple nodes. When a component performance changes, the health indicator change its color, so developers can fix the problem immediately, before affecting the whole application and users complain.

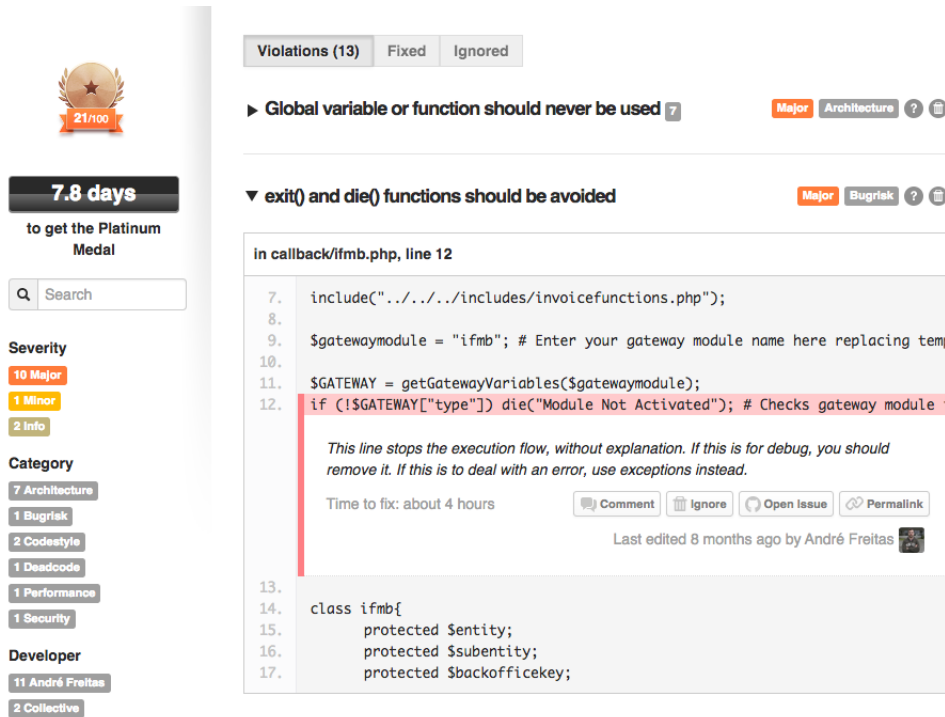
2.5.3 SensioLabsInsight

SensioLabsInsight in figure 2.6, is a web service ¹⁸ that continuously analyzes PHP projects in terms of security, bugs and other quality checks. It also does dynamic analysis to improve di-

¹⁸<https://insight.sensiolabs.com/>

State of the art

agnostic accuracy. Integrates with Github and Bitbucket services and is free for Open Source projects.



The screenshot displays the SensioLabsInsight dashboard. On the left, there is a sidebar with a progress indicator showing '21/100' and a goal of '7.8 days to get the Platinum Medal'. Below this are filters for 'Severity' (10 Major, 1 Minor, 2 Info), 'Category' (7 Architecture, 1 Bugriak, 2 Codestyle, 1 Deadcode, 1 Performance, 1 Security), and 'Developer' (11 André Freitas, 2 Collective). The main area shows a list of violations. The first violation is 'Global variable or function should never be used' (Major, Architecture). The second violation is 'exit() and die() functions should be avoided' (Major, Bugriak). The selected violation shows a code snippet from 'callback/ifmb.php, line 12' with the following code:

```
7. include("../../includes/invoicefunctions.php");
8.
9. $gatewaymodule = "ifmb"; # Enter your gateway module name here replacing temp
10.
11. $GATEWAY = getGatewayVariables($gatewaymodule);
12. if (!$GATEWAY["type"]) die("Module Not Activated"); # Checks gateway module
```

A red box highlights line 12, with a message: 'This line stops the execution flow, without explanation. If this is for debug, you should remove it. If this is to deal with an error, use exceptions instead.' Below the code, it indicates 'Time to fix: about 4 hours' and provides buttons for 'Comment', 'Ignore', 'Open Issue', and 'Permalink'. The violation was last edited 8 months ago by André Freitas.

Figure 2.6: Example of a report from SensioLabsInsight

2.5.4 Code Climate

Code Climate ¹⁹ in figure 2.7, is another service that analyzes PHP, Python, Javascript and Ruby using static analysis. It essentially produce warnings about issues in code complexity, duplication, style and readability. Also displays insights about churn (lines changed) versus code quality. The dashboard is very complete since we can check listed issues or inspect code along with the warnings produced. It has a free plan for Open Source projects.

¹⁹<https://codeclimate.com/>

State of the art

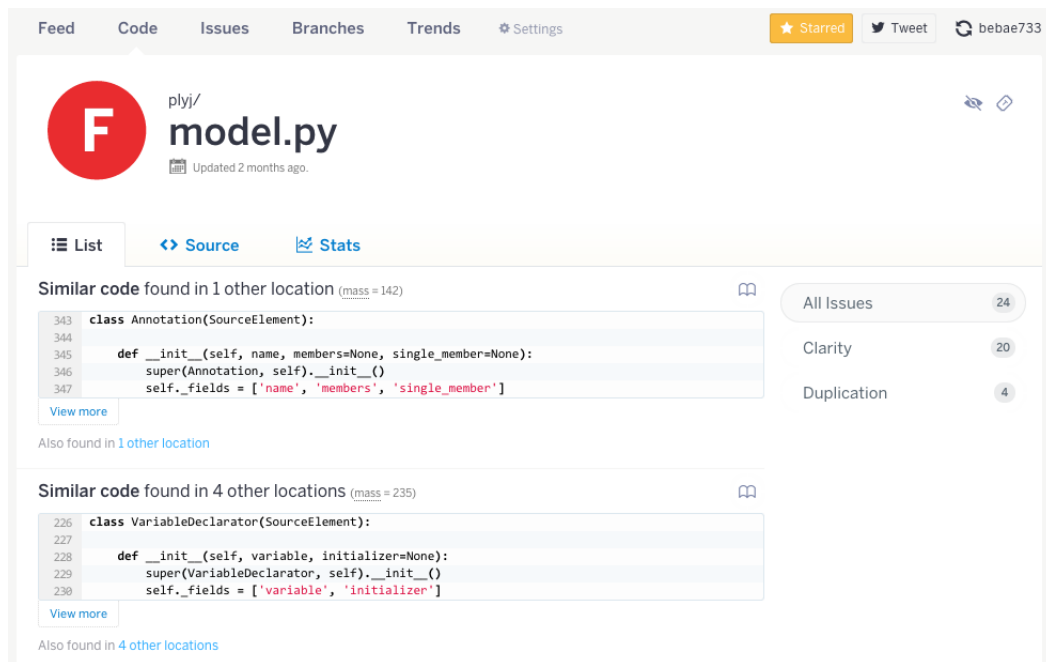


Figure 2.7: Issues listed on Code Climate

2.5.5 Pull Review

Pull Review²⁰ in figure 2.8, is an automatic code review service (static analysis) just for Ruby. It gives feedback for style, duplication, code smells, documentation, security and tests. It has the ability of linking a Github, Gitlab or Bitbucket repository and is also free for Open Source.

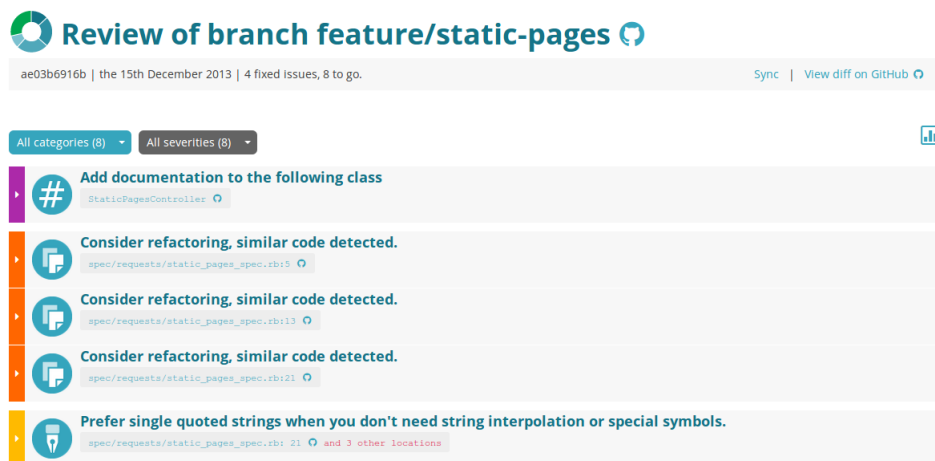


Figure 2.8: Pull Review report

²⁰<https://pullreview.com/>

Chapter 3

A Technique to Estimate Defect Probabilities

In this chapter it is discussed the methodologies used to answer the research questions.

3.1 Schwa

Schwa is the tool developed for research. Source code is available at Github¹ as an Open Source project under MIT 2.0 license. It was developed in Python and is hosted at PyPi², the Python Package Index.

3.1.1 Installation

Schwa relies on Python 3 and Git and they are the only dependencies. Can be easily installed using pip³.

```
1 pip3 install schwa --pre
```

Listing 3.1: Schwa installation command

3.1.2 Usage

It can be used as a command line tool and imported as a Python package. An example of running Schwa, is analyzing the last 20 commits of the Joda Time repository:

¹<https://github.com/andrefreitas/schwa>

²<https://pypi.python.org/pypi/Schwa>

³<https://pip.pypa.io/en/latest/installing.html>

```
1 schwa git/joda-time --commits 20
```

Listing 3.2: Running Schwa on Joda Time

3.1.3 Visualization

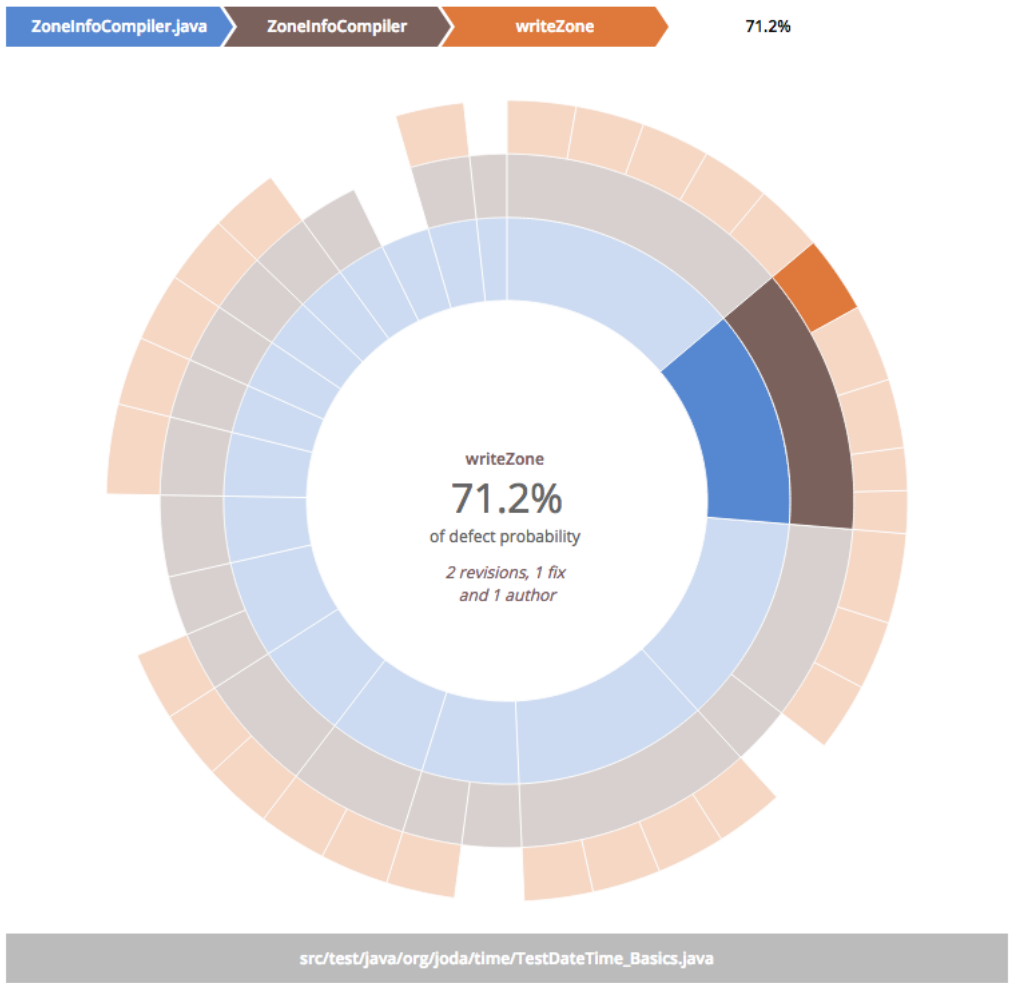


Figure 3.1: Sunburst results for Joda Time

The report is launched in a browser and is displayed in a Sunburst chart, described in Figure 3.1, that is a simple way of showing a hierarchy of results. It is inspired from the visualization style of Crowbar. Granularity of components increases from the center to the periphery and the longer the arc, the higher the probability of defect. Blue nodes are files, brown are classes and orange are methods. The path of selected files are displayed in the bottom.

This visualization also gives some reasoning about the defect probability, displaying the values for revisions, fixes and authors.

3.1.4 Process

Mining a Software repository can be essentially divided in two phases: Extraction and Analysis. The rationale is that first we only extract the most important information and then extracted data is analyzed. By using a generic representation of a repository as the input for the analysis, Schwa can be extended with more SCM tools, such as Mercurial and Apache Subversion. The process pipeline is described in figure 3.2.



Figure 3.2: Process pipeline

3.1.4.1 Extraction

Extraction is the phase that takes more time due the amount of I/O operations reading blobs, even with code parallelization. Considering these performance issues, Schwa iterates over commits, so it is possible to just extract for example, the last 10 commits. For each commit every change is parsed so the most important information is:

- **Message** Commit message that is used to evaluate if the commit is fixing a bug;
- **Author** Commit author's email to track the number of contributors a component had;
- **Timestamp** An integer with the Unix Timestamp, used to track changes in components and achieving time relevance (TWR), that is distinguishing recently changed components from old components;
- **Diffs** The list of commit changes in components, such as files, classes and methods.

Schwa extracts data from Git repositories using the GitPython library⁴. For Java source files we track changes until the method granularity. For other programming languages we only track changes until the file granularity.

Parsing Java is possible by using the Plyj⁵ library, that has been modified to include line information for Classes and Methods in the Abstract Syntax Tree.

When the extraction finishes, all the important information has been collected and is represented in the model of figure 3.3.

⁴<https://github.com/gitpython-developers/GitPython>

⁵<https://github.com/musiKk/plyj>

A Technique to Estimate Defect Probabilities

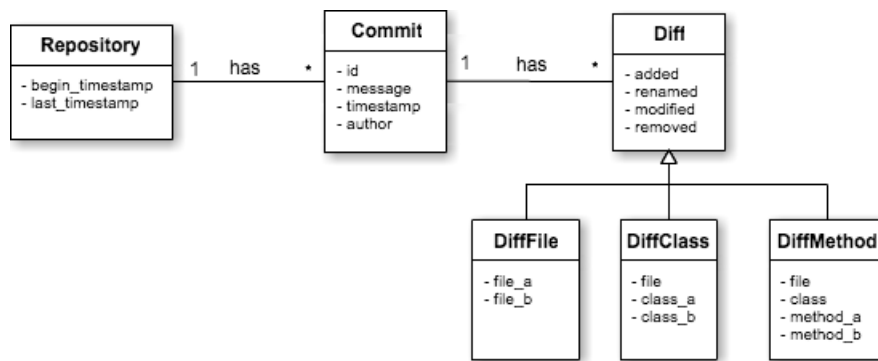


Figure 3.3: Repository Model

Begin and last timestamps in the Repository class are referring to the first and last commits and are necessary for the TWR formula. Diffs have been divided in subclasses for each granularity. A diff is a result of changing a version A resulting in another version B. To make more clear this concept, consider the following patch:

```
1 class API {
2     static void login(String email, String password,
3 AsyncHttpResponseHandler responseHandler){
4         RequestParams params = new RequestParams();
5         params.put("email", email);
6         params.put("password", password);
7 +     params.put("token", "j76g367f4");
8         client.post(url + "/login", params, responseHandler);
9     }
10
11 + static void getShows(AsyncHttpResponseHandler responseHandler) {
12 +     client.get(url + "/shows", responseHandler);
13 + }
14
15 - static void getUsers(AsyncHttpResponseHandler responseHandler) {
16 -     client.get(url + "/users", responseHandler);
17 - }
18 }
```

Listing 3.3: Patch in API.java

The list of diffs instances would be:

```
1 DiffFile(file_a="API.java", file_b="API.java", modified=True)
2 DiffClass(file_name="API.java", class_a="API", class_b="API", modified=True)
3 DiffMethod("API.java", class_name="API", method_a="login", method_b="login",
4     modified=True)
4 DiffMethod("API.java", class_name="API", method_b="getShows", added=True)
```

```
5 DiffMethod("API.java", class_name="API", method_a="getUsers", removed=True)
```

Listing 3.4: List of Diff instances

We are currently supporting only renaming of files since detecting a renaming of a Class or Method can be subjective. For example, it is not trivial by parsing a patch, recognizing the difference of deleting and creating a new method with the same body or only changing the method signature.

3.1.4.2 Analysis

Analysis receives a Repository (figure 3.3) as the input and will iterate again over commits but now with the objective of tracking the following metrics:

- **Revisions** The more a component had recently new changes, the higher will be revisions score. Graves *et al.* showed that revisions is a predictive variable [GKMS00];
- **Fixes** The more a component had recently new bug-fixes, the higher will be fixes score; Zimmerman *et al.* showed that past defects have the highest correlation with future defects [ZPZ07];
- **Authors** The more a component had recently new authors, the higher will be authors score. Authors is a process metric that can be also used to predict defects [MPS08, DLR12].

Metrics are tracked the same way for any granularity and they were selected after a revision of the state of the art in MSR. Considering our goal of improving Crowbar fault localization results, the Time-Weighted Risk approach is the one that fits this scenario, since it is a rank-based technique. TWR is not only used only for Fixes but also for Revisions and Authors, since they can benefit of time relevance, something that was not possible when using counters.

We also found that the TWR curve in the Figure 2.1, should be modified, because even recently changed components were having low score. The problem, is that when $t_i \leq 0.8$, TWR is low. We want the curve to start growing sooner so therefore we modified the TWR equation with the Time Range (TR) parameter, resulting in a new expression:

$$twr(t_i) = \frac{1}{1 + e^{-12t_i + 2 + ((1-TR)*10)}} \quad (3.1)$$

TR varies from 0 to 1 . In figure 3.4 with TR = 0.5, the curve goes to the left giving more score to older timestamps and increasing the score of $t_i = 1.0$ to 1.0.

A Technique to Estimate Defect Probabilities

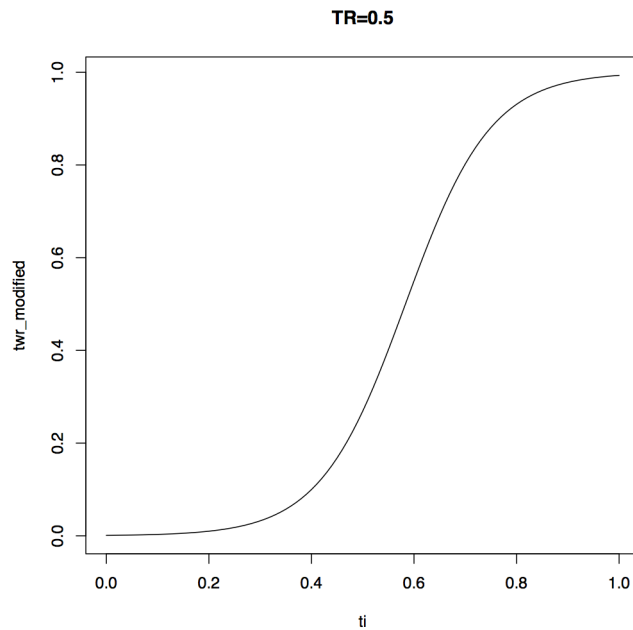


Figure 3.4: Modified Time Weighted Risk (TWR)

With a deeper understanding about how TWR works, algorithm 1 describes the steps involved in the analysis, in a simplified way.

```
for each commit in the repository do
  twr = compute twr
  for each component in the commit do
    component.revisions += twr
    if commit is bug fix then
      | component.fixes += twr
    end
    if is new author then
      | component.authors += twr
    end
  end
end
```

Algorithm 1: Analysis algorithm

We detect if a commit is fixing a bug by searching for keywords in the message. Since Github⁶ is the most used Git repository hosting service, the regular expression was based on Github's syntax for closing issues by commit messages⁷. The regular expression is the following:

⁶<http://github.com>

⁷<https://help.github.com/articles/closing-issues-via-commit-messages/>

```
1 fix(e[ds])?|bugs?|defects?|patch|close([sd])?|resolve([sd])?
```

Listing 3.5: Bug-fixing message regular expression

3.1.4.3 Defect prediction computation

When the analysis is finished, Schwa computes the defect probability for every component by doing a weighted average of the metrics:

$$score = revisions * revisions_{weight} + fixes * fixes_{weight} + authors * authors_{weight} \quad (3.2)$$

Then the score is normalized to a value from 0 to 1, that is the estimation of the defect probability:

$$defect_{probability} = 1 - e^{-score} \quad (3.3)$$

By default, Schwa uses the weights: 0.25 for revisions, 0.25 for authors and 0.5 for fixes. These weights are based from the existing research in MSR [GKMS00, ZPZ07, MPS08, DLR12]. They may be different for each type of software project. For example, in a project with a lot of contributors, tracking authors can be important, but in a project with just a contributor, this metric is not relevant because the number of authors never changes.

3.2 Features weight estimation

Estimating the weights of each feature: revisions, fixes and authors is an optimization problem. Our approach was using Genetic Algorithms for searching the best solution and encoding individuals as:

$$\{R, F, A\} \quad (3.4)$$

R , F and A are the weights from 0 to 1 for revisions, fixes and authors, respectively. They have been encoded to binary with 3 bits of precision, that means that we can represent $2^3 = 8$ different values for each feature. Increasing bits precision has the cost of decreasing performance.

The fitness function 3.5 is the sum of the distance between involved and not involved components for every bug-fixing commit m with a penalty for invalid solutions: weights cannot be 0 and their sum must be 1. This rationale results in the following expressions:

$$fitness(R, F, A) = \begin{cases} \sum_{m \in commits_{bug\ fixing}} distance(R, F, A, m) & \text{if constraints}(R, F, A) \\ -\infty & \text{otherwise} \end{cases} \quad (3.5)$$

$$constraints(R, F, A) = (R + F + A) == 1 \wedge (R * F * A) > 0 \quad (3.6)$$

$$distance(R, F, A, m) = \frac{\sum_{c \in involved(m)} score(R, F, A, c)}{|involved(m)|} - \frac{\sum_{c \in (Components \setminus involved(m))} score(R, F, A, c)}{|(Components \setminus involved(m))|} \quad (3.7)$$

$$score(R, F, A, c) = c_{revisions} * R + c_{fixes} * F + c_{authors} * A \quad (3.8)$$

This means that in bug-introducing commits, faulty components should have higher score than non faulty components. Besides finding features weights, this approach helps us validate if Schwa is predicting defects correctly. Due to performance constraints, the fitness function only evaluates components at the file granularity. Note that this granularity is only for Schwa learning mode. For defect prediction, Schwa still operates with method granularity for Java.

3.3 Diagnostic cost

Crowbar outputs a rank of components ordered by their probability of having the fault. To measure the quality of this diagnostic, the heuristic used is Diagnostic Cost [CAFd13] that is the average of the minimum and maximum distance that faulty components are from the top of the rank. For multiple faults, this cost is computed considering the lowest faulty component.

$$diagnostic_{cost} = \frac{min_{distance} + max_{distance} - |faulty|}{2} \quad (3.9)$$

3.3.1 Example

C_2 is the component that have the fault.

Position	Rank 1	Rank 2
1	$C_3(0.234)$	$*C_2(0.500)$
2	$C_5(0.145)$	$C_5(0.120)$
3	$*C_2(0.145)$	$C_3(0.120)$
Diagnostic Cost	1	0

Table 3.1: Examples of ranks

In Rank 1 C_5 and C_2 components have the same probability so C_2 position can be 2 or 3 depending on the result of ordering the rank. In Rank 2 C_2 is on the first position so the cost is 0.

3.3.2 Schwa integration with Crowbar

By using Schwa, the goal is improving Barinel results, reducing the Diagnostic Cost by evaluating if defect probabilities should be used in priors (p_j), goodness (g_j) or both parameters. In Barinel the meaning of these parameters is the following:

Prior

The probability a component have of causing an error and is by default $\frac{1}{1000}$, based on the heuristic that for every 1000 lines of code heuristic there is one bug.

Goodness

The probability of a component behaving normally and by default is computed by using the Maximum Likelihood Estimation that is heavy to compute. By default is 0.5.

If the defect probability of Schwa is used for goodness, it is computed by:

$$\Pr(obs_i, e_i | d) = 1 - defect_{probability}(c) \quad (3.10)$$

A Technique to Estimate Defect Probabilities

Chapter 4

Experimental results

This chapter provides the results of the experiments conducted. The setup and configurations are also described. There is a public page with the experimental results of Schwa on Github ¹.

4.1 Features weight estimation

In this section we present the results of learning the features weights, that is the importance of each of them, when computing the defect probability.

4.1.1 Experimental setup

To run this experiment, Schwa was invoked in the learning mode with 5, 50 and 100 commits with the script in listing 4.1. The version of Schwa used was 0.1.dev24(tagged on Git).

Running genetic algorithms takes a substantial time of computation, so we have used Crowdsourcing to run the experiment in a variety of projects. We collected data from academic, enterprise and Open Source projects to have results from different contexts.

```
1 if [ -z "$1" ]
2 then
3   echo "usage: $0 <git repository path>"
4 else
5   touch report_${USER}.txt
6   echo "This will take a while..."
7   echo "Learning with 5 commits"
8   schwa $1 --commits 5 -l >> report_${USER}.txt
9   echo "Learning with 50 commits"
10  schwa $1 --commits 50 -l >> report_${USER}.txt
11  echo "Learning with 100 commits"
```

¹<https://github.com/andrefreitas/schwa/wiki/Experiments>

Experimental results

```
12 schwa $1 --commits 100 -l >> report_${USER}.txt
13 echo "Thank you! You are the best! Send report_${USER}.txt to Andre :)"
14 fi
```

Listing 4.1: Shell script used to learn features weights

4.1.2 Results

Here we show the results for each repository. In the following tables, in each row we have the maximum number of commits, the weights of revisions, fixes and authors and the fitness value. The main objective of these tables is providing the importance of each feature for a certain project.

4.1.2.1 Schwa

The experiment was run in Schwa² itself, that have only one contributor. It is developed mostly with Python along with HTML, CSS and Javascript.

Commits	Revisions	Fixes	Authors	Fitness
5	0.2857	0.2857	0.4286	0
50	0.7143	0.1429	0.1429	1.0699
100	0.7143	0.1429	0.1429	1.1875

Table 4.1: Schwa results

For 5 commits, the fitness function is 0, so it is not possible to estimate correctly the weights. For 50 and 100 commits, revisions is the most important feature.

4.1.2.2 Libcrowbar

Libcrowbar is the main repository of Crowbar and have multiple academic contributors. The technologies used are Java, C++, HTML, CSS and Javascript.

Commits	Revisions	Fixes	Authors	Fitness
5	0.1429	0.1429	0.7143	0.3486
50	0.7143	0.1429	0.1429	1.3639
100	0.7143	0.1429	0.1429	0.3387

Table 4.2: Libcrowbar results

For the last 5 commits, authors is the most important feature and for 50 and 100 commits, is revisions.

²<https://github.com/andrefreitas/schwa>

Experimental results

4.1.2.3 Joda Time

Joda Time³ is an Open Source time library for Java with multiple contributors.

Commits	Revisions	Fixes	Authors	Fitness
5	0.1429	0.2857	0.5714	0
50	0.1429	0.7143	0.1429	-2.1874
100	0.1429	0.7143	0.1429	0.3893

Table 4.3: Joda Time results

For this project only for 100 commits the fitness function is > 0 , showing that fixes is the most important feature.

4.1.2.4 Meo Arena

Meo Arena⁴ is a mobile application developed for Android in an academic context with two contributors. It relies on Java and Python.

Commits	Revisions	Fixes	Authors	Fitness
5	0.2857	0.4286	0.2857	0
50	0.7143	0.1429	0.1429	1.6712
100	0.7143	0.1429	0.1429	0.7368

Table 4.4: Meo Arena results

For 5 commits, fitness is equal to zero. For 50 and 100 commits the most important feature is revisions.

4.1.2.5 Mongo Java Driver

Mongo Java driver⁵ is an Open Source projects and enables Java applications to communicate with a MongoDB server. It is developed with Java and Groovy and has multiple contributors.

Commits	Revisions	Fixes	Authors	Fitness
5	0.7143	0.1429	0.1429	0.3486
50	0.7143	0.1429	0.1429	0.1685
100	0.1429	0.7143	0.1429	1.4666

Table 4.5: Mongo Java driver results

For 5 and 50 commits the most important feature is revisions but for 100, is fixes.

³<https://github.com/JodaOrg/joda-time>

⁴<https://bitbucket.org/andrefreitas/feup-cmov-meoarena/>

⁵<https://github.com/mongodb/mongo-java-driver>

Experimental results

4.1.2.6 Scraim

Scraim⁶ is a web-based project management tool developed by the company Strongstep. It is build on top of Redmine and uses Ruby on Rails.

Commits	Revisions	Fixes	Authors	Fitness
5	0.4286	0.1429	0.4286	0.3486
50	0.1429	0.1429	0.7143	0.3146
100	0.1429	0.7143	0.1429	0.9399

Table 4.6: Scraim results

For 5 commits, revisions and authors are both important. For 50, authors is the most important and for 100 is fixes.

4.1.2.7 Trainsim

Trainsim is an academic project developed with Java and Swing with one contributor.

Commits	Revisions	Fixes	Authors	Fitness
5	0.1429	0.7143	0.1429	0
50	0.7143	0.1429	0.1429	1.5945
100	0.5714	0.2857	0.1429	2.0425

Table 4.7: Trainsim results

For 5 commits fitness is zero, for 50 and 100 the most important is revisions. Fixes importance increased from 50 to 100 commits.

4.1.2.8 ShiftForward

ShifForward⁷ is a company that develops advertising technology with Scala. They have contributed with results of three projects.

Commits	Revisions	Fixes	Authors	Fitness
5	0.4286	0.1429	0.4286	0
50	0.4286	0.4286	0.1429	1.9044
100	0.2857	0.4286	0.2857	3.3134

Table 4.8: Adstax results

In the Adstax project, for 5 commits the fitness is zero. For 50 commits, revisions and fixes are both important and for 100, fixes is the most important.

⁶<https://scraim.com/>

⁷<http://shiftforward.eu>

Experimental results

Commits	Revisions	Fixes	Authors	Fitness
5	0.2857	0.1429	0.5714	0.4408
50	0.1429	0.1429	0.7143	0.4457
100	0.1429	0.7143	0.1429	-1.4159

Table 4.9: Boxer results

In the Boxer project, for 5 and 50 commits, authors is the most important feature. For 100 commits, the fitness is negative.

Commits	Revisions	Fixes	Authors	Fitness
5	0.7143	0.1429	0.1429	0.9439
50	0.1429	0.7143	0.1429	1.9547
100	0.1429	0.4286	0.4286	1.3927

Table 4.10: Apso results

In the project Apso, revisions is the most important feature for 5 commits. For 50 commits is fixes and for 100 commits is fixes and authors.

4.1.2.9 Hivedb

Hivedb⁸ is an Open Source framework for horizontally partitioning MySQL systems. It is developed mostly in Java with multiple contributors.

Commits	Revisions	Fixes	Authors	Fitness
5	0.7143	0.1429	0.1429	0
50	0.1429	0.7143	0.1429	0.1739
100	0.7143	0.1429	0.1429	0.9095

Table 4.11: Hivedb results

For 5 commits the fitness function is zero. For 50 commits the most important feature is fixes and for 100, is revisions.

4.1.2.10 Teamengine

Teamengine⁹ is an Open Source engine to test web services and other resources in Java.

Commits	Revisions	Fixes	Authors	Fitness
5	0.7143	0.1429	0.1429	0
50	0.1429	0.7143	0.1429	1.1356
100	0.7143	0.1429	0.1429	3.1080

Table 4.12: Teamengine results

⁸<http://hivedb.org>

⁹<https://github.com/opengeospatial/teamengine>

Experimental results

For 5 commits the fitness function is 0. Fixes is the most important feature for 50 commits and for 100 is revisions.

4.1.2.11 Automatalib

Automatalib¹⁰ is an Open Source Java Library for representing automata, graphs and transition systems.

Commits	Revisions	Fixes	Authors	Fitness
5	0.1429	0.1429	0.7143	0.3486
50	0.1429	0.7143	0.1429	-0.1952
100	0.4286	0.4286	0.1429	0.5261

Table 4.13: Automatalib results

For 5 commits, authors is the most important feature. For 50 commits the fitness is negative. Revisions and fixes have the same weight for 100 commits.

4.1.2.12 CDI TCK

CDI TCK¹¹ is an Open Source Context and Dependency Injection for Java EE developed with Java.

Commits	Revisions	Fixes	Authors	Fitness
5	0.1429	0.5714	0.2857	0
50	0.2857	0.2857	0.4286	0
100	0.1429	0.7143	0.1429	-0.3247

Table 4.14: CDI TCK results

For 5 and 50 commits fitness is zero and for 100 is negative.

4.2 Diagnostic cost

In this section, the results of computing the diagnostic cost for each configuration of Schwa in Crowbar are presented. In this experiment, the computational cost is also substantial and crowd-sourcing could not be used since we need to inspect the behaviour of Crowbar with Schwa. Considering that these experiments were run in a laptop (can last hours) with the constraints of finding Java projects that use Git, have tests and support Maven, this phase took weeks to find conclusive results.

Joda Time and CDI TCK projects were selected to run these experiment since they have a Git repository and are Java projects with tests, so they both can be used with Crowbar.

¹⁰<https://github.com/misberner/automatalib>

¹¹<https://github.com/cdi-spec/cdi-tck>

4.2.1 Experimental setup

For each project we had setup the experimental environment with the following steps:

- Compute the weights for revisions, fixes and authors with Schwa learning mode;
- Create a .schwa.yml in the root of the repository with the weights and maximum commits;
- Insert bugs (e.g. wrong comparison) in methods and commit the changes;
- Evaluate the diagnostic cost for using Schwa with priors, goodnesses or both.

4.2.2 Results

The results are presented with the history of commits and configurations of Schwa.

4.2.2.1 Joda Time

The sequence of commits applied in Joda Time is available on table 4.15 along with the commits that inserted bugs.

Order	Commit	Description
1	8207a55	Added a defect in DateTime.java in withZoneRetainfields()
2	74149c0	Added a defect in Duration.java in minus()
3	22a5f71	Fixed withZoneRetainfields() bug
4	0945c34	Fixed minus() bug and added another bug
5	92adf94	Fixed previous bug and added one in getMaximumValue()

Table 4.15: Commits applied to Joda Time

```

1  public DateTime withZoneRetainFields(DateTimeZone newZone) {
2      newZone = DateTimeUtils.getZone(newZone);
3      DateTimeZone originalZone = DateTimeUtils.getZone(getZone());
4  -     if (newZone == originalZone) {
5  +     if (newZone != originalZone) {
6          return this;
7      }

```

Listing 4.2: Commit 8207a55 patch

```

1  public Duration minus(long amount) {
2  -     return withDurationAdded(amount, -1);
3  +     return withDurationAdded(amount, -2);
4      }
5
6  public Duration minus(ReadableDuration amount) {

```

Experimental results

```
7 -     if (amount == null) {
8 +     if (amount != null) {
9         return this;
10    }
11    return withDurationAdded(amount.getMillis(), -1);
12 }
```

Listing 4.3: Commit 74149c0 patch

```
1 final class BasicDayOfMonthDateTimeField extends PreciseDurationDateTimeField {
2     int month = values[i];
3     for (int j = 0; j < size; j++) {
4         if (partial.getFieldType(j) == DateTimeFieldType.year()) {
5 -         int year = values[j];
6 +         int year = values[i];
7         return iChronology.getDaysInYearMonth(year, month);
8     }
9 }
```

Listing 4.4: Commit 92adf94 patch

The configurations used are in table 4.16. Schwa was used with the default configuration values in configuration #1. In the configuration #2 the weights learned from the genetic algorithms are used along with the time range($tr=0.6$), to increase the defect probabilities of the last changed components.

Configuration	Commits	Revisions	Fixes	Authors	Time Range
#1	20	0.25	0.5	0.25	0
#2	5	0.15	0.7	0.15	0.6

Table 4.16: Schwa configurations for Joda Time

The results for the diagnostic cost are presented in table 4.17 and they are grouped in scenarios. A scenario is a unique configuration and revision (commit).

Experimental results

Schwa	Commit	Configuration	Diagnostic Cost
First scenario			
None	74149c0	#1	0
Both	74149c0	#1	1
Priors	74149c0	#1	1
Goodnesses	74149c0	#1	1
Second scenario			
None	22a5f71	#1	0
Both	22a5f71	#1	0
Priors	22a5f71	#1	1
Goodnesses	22a5f71	#1	0
Third scenario			
None	92adf94	#2	20
Both	92adf94	#2	21
Priors	92adf94	#2	20
Goodnesses	92adf94	#2	21

Table 4.17: Diagnostic cost for Joda Time

As presented in table 4.17, using configuration #1 in the first scenario, when Schwa is used the diagnostic cost increases to 1. In the second scenario, by fixing one of the bugs, the diagnostic cost only increases when Schwa is used for priors. In the third scenario, by fixing the previous bugs, add another one and using the time range parameter (new configuration), the diagnostic cost increases for Both and Goodnesses, but is the same for Priors.

For the third scenario, we also evaluated what would be the maximum diagnostic cost if Schwa would give optimal results. The defect probability of the faulty component was set to 0.9, and for the healthy components was set to 0.1 (note that this probabilities are only for components involved in the last commits analyzed by Schwa). By evaluating the rank again, the diagnostic cost was 20 for priors and 21 for goodnesses and both.

4.2.2.2 CDI TCK

The commit applied in CDI TCK is on Table 4.18.

Commit	Description
1bc2e2d	Added a defect in ActionSequence.java toString() method

Table 4.18: Commits applied to CDI TCK

```

1  @Override
2  public String toString() {
3  -     return String.format("ActionSequence [name=%s, data=%s]", name, getData());
4  +     return String.format("ActionSequenc [name=%s, data=%s]", name, getData());
5  }

```

Experimental results

Listing 4.5: Commit 1bc2e2d patch

Configuration	Commits	Revisions	Fixes	Authors	Time Range
#1	20	0.15	0.7	0.15	0
#2	20	0.7	0.15	0.15	0
#3	20	1	0	0	0
#4	5	1	0	0	0.6
#5	5	0.15	0.7	0.15	0.6

Table 4.19: Schwa configurations for CDI TCK

The results for the diagnostic cost are the following:

Schwa	Commit	Configuration	Diagnostic Cost
First scenario			
None	1bc2e2d	#1	0
Both	1bc2e2d	#1	6
Priors	1bc2e2d	#1	8
Goodnesses	1bc2e2d	#1	0
Second scenario			
Both	1bc2e2d	#2	6
Priors	1bc2e2d	#2	7
Goodnesses	1bc2e2d	#2	0
Third scenario			
Both	1bc2e2d	#3	0
Priors	1bc2e2d	#3	7
Goodnesses	1bc2e2d	#3	0
Fourth scenario			
Both	1bc2e2d	#4	0
Priors	1bc2e2d	#4	9
Goodnesses	1bc2e2d	#4	0
Fifth scenario			
Both	1bc2e2d	#5	9
Priors	1bc2e2d	#5	1
Goodnesses	1bc2e2d	#5	1

Table 4.20: Diagnostic cost for CDI TCK

The experiment in CDI TCK was conducted by trying a variety of configurations, to evaluate their impact. For the applied patch, the diagnostic cost is zero without the usage of Schwa. In the first scenario by giving more importance to fixes, the diagnostic cost is worse for all options, except for goodnesses. In the second scenario, by giving more importance to revisions, the results are practically the same: worse for all options except goodnesses.

Experimental results

In the third and fourth scenarios, by giving only importance to revisions the diagnostic cost is zero for goodneses and both. The time range parameter was changed in the fourth scenario.

In the fifth scenario, by combining the usage of time range and giving 0.15 for revisions and authors and 0.7 for fixes, the diagnostic cost increased in all options.

Experimental results

Chapter 5

Discussion

In this chapter it is discussed findings and conclusions relative to the initial research questions.

5.1 Features weight estimation

The initial goal was finding a way of generalizing the weights of each features. Although, since every software project is different, we found that it depends on the project:

Features weights are different for each project

In Schwa that is a project with only contributor, the weights for 50 and 100 commits were consistent: revisions is the most important feature. But, for Joda Time with 100 commits, the most important was fixes.

Noise on tracking fixes

Authors and revisions are the only features that are measured with accuracy. Fixes are tracked based on bug-fixing commits that have noise and have an impact on defect predictions results, a problem that is discussed in MSR research[[HJZ12](#)].

Precision of Genetic Algorithms

We represented individuals with 3 bits of precisions at the cost of performance. With more computational power (e.g. cluster) we could increase precision to see if we would find different results.

Configurable features weights

Since features could not be generalized, we introduced a new feature to Schwa: a configuration file `.schwa.yml`, to allow developers to change the weights of revisions, fixes and authors. With this, developers can run Schwa in learning mode first and configure it with the weights learned.

5.2 Diagnostic cost

The results from diagnostic cost experiments indicate that we could not improve the results of Crowbar but found an alternative way of estimating defect probabilities in the Barinel technique:

Improvement of diagnostic results

We could not find an example of Schwa improving the diagnostic results of Crowbar. But, we must note that even with optimal defect predictions results from Schwa, in some cases the diagnostic cost cannot be improved, as seen in Joda Time.

Importance of recently changed components

In the first results from Joda Time we were getting worse results because faulty components that had been recently changed, had low defect probability. By modifying the TWR function with the Time Range parameter, when Schwa was used in priors, it did not get worse results.

Faster defect prediction results with Schwa

Since the Barinel algorithm uses the MLE to estimate goodnesses and priors, this process can take for example 2 hours in some cases. By using Schwa, we reduced this phase to less than 1 minute.

Computational power

A cluster is better suited than a laptop to get results in a more convenient time. Schwa is I/O intensive because it is parsing and extracting code from commits. Crowbar have a substantial time complexity by running the MLE algorithm and can benefit of faster CPUs.

5.3 Threats to validity

Regarding the experiments for estimating features weight, the usage of 3 bits for representing the weights of individuals can limit the possibility of searching better solutions. For the diagnostic cost, the results are just from two projects that are open source.

Chapter 6

Conclusions and Further Work

We have developed a framework capable of predicting software defects from repositories, with a web-based graphical report. The creation of a learning mode for Schwa with genetic algorithms, gives researchers the ability of evaluating new features to extract from repositories, making Schwa a convenient framework to study Mining Software Repositories.

Schwa should be combined with other techniques, since it is not completely accurate. Code review is an example of an activity that can benefit from this tool, allowing developers to focus in the most important components.

The usage of Python allowed a fast prototyping of ideas due its simplicity and the existing of useful libraries. Mining Software Repositories is a time-consuming activity so research in this subject can benefit from the usage of clusters.

6.1 Goals satisfaction

We successfully created a defect prediction technique based on MSR approaches capable of learning features, until the method granularity for Java projects. Our initial goal of generalizing features weights was refuted by the experimental results, that shown that for each projects they are different.

Although we did not improve the accuracy of Barinel, we have come with an alternative technique of computing defect probabilities in less time. For example, since Barinel for Joda Time can take 2 hours to run MLE, now with Schwa, this phase takes less that 1 minute, so it is a substantial achievement.

6.2 Further work

The technique used in Schwa for learning features can be improved with optimizations in the binary representation and code parallelization. There are plenty of improvements that can be done

Conclusions and Further Work

in Schwa:

- Support of more programming languages;
- Improve performance on extraction by developing a Python module in C;
- Add charts for revisions, fixes and authors evolution in the visualization, to support the results with more reasoning;
- Develop a SaaS platform for Schwa, similar to Codeclimate and Codacy.

MSR research could benefit of new techniques that reduce noise in the classification of bug-fixing commits, that can exploit issue trackers. Schwa could benefit from reducing this noise.

With more computational power, we could evaluate with more examples, the gain of using Schwa in Crowbar, by finding an example where the diagnostic cost decreased.

References

- [AZG09] Rui Abreu, Peter Zoetewij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09*, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society.
- [CAFd13] J. Campos, R. Abreu, G. Fraser, and M. d’Amorim. Entropy-based test generation for improved fault localization. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 257–267, Nov 2013.
- [Car13] Emil Carlsson. Mining git repositories : An introduction to repository mining, 2013. Linnaeus University, Department of Computer Science. Degree of Bachelor.
- [CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, Jun 1994.
- [CRPA12] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. GZoltar: an Eclipse plug-in for Testing and Debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012*, pages 378–381, New York, NY, USA, 2012. ACM.
- [DLR12] Marco D’Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, August 2012.
- [FN99] N.E. Fenton and M. Neil. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5):675–689, Sep 1999.
- [GKMS00] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, 26(7):653–661, Jul 2000.
- [HJZ12] Kim Herzig, Sascha Just, and Andreas Zeller. It’s not a bug, it’s a feature: How misclassification impacts bug prediction. Technical report, Universität des Saarlandes, Saarbrücken, Germany, August 2012.
- [HNB⁺13] Hadi Hemmati, Sarah Nadi, Olga Baysal, Oleksii Kononenko, Wei Wang, Reid Holmes, and Michael W. Godfrey. The MSR cookbook: Mining a decade of research. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13*, pages 343–352, Piscataway, NJ, USA, 2013. IEEE Press.
- [ISO11] ISO. Systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models. ISO ISO/IEC

REFERENCES

- 25010:2011, International Organization for Standardization, Geneva, Switzerland, 2011.
- [JH05] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 273–282, November 2005.
- [KWZ08] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, March 2008.
- [KZWJZ07] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering, ICSE '07*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.
- [LLS⁺13] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 372–381, Piscataway, NJ, USA, 2013. IEEE Press.
- [MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 181–190, New York, NY, USA, 2008. ACM.
- [MS08] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08*, pages 128–137, Washington, DC, USA, 2008. IEEE Computer Society.
- [PAW14] Alexandre Perez, Rui Abreu, and Eric Wong. A survey on fault localization techniques. 2014. Technical report.
- [Ser13] F. Servant. Supporting bug investigation using history analysis. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 754–757, Nov 2013.
- [SJ12] Francisco Servant and James A. Jones. History slicing: Assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12*, pages 43:1–43:11, New York, NY, USA, 2012. ACM.
- [SLL⁺11] Caitlin Sadowski, Chris Lewis, Zhongpeng Lin, Xiaoyan Zhu, and E. James Whitehead, Jr. An empirical analysis of the fixcache algorithm. In *Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11*, pages 219–222, New York, NY, USA, 2011. ACM.
- [WD09] W. Eric Wong and Vidroha Debroy. A survey of software fault localization, 2009. Technical report.

REFERENCES

- [WH05] Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.*, 31(6):466–480, June 2005.
- [ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society.