FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Software Repository Mining for Estimating Software Component Reliability

André Duarte

WORKING VERSION

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rui Maranhão

Second Supervisor: Alexandre Perez

June 24, 2016

Software Repository Mining for Estimating Software Component Reliability

André Duarte

Mestrado Integrado em Engenharia Informática e Computação

June 24, 2016

Abstract

Given the rising necessity of identifying errors on the source code of software, in order to make the developers work easier and to speed up the development process, many progresses have been made in its automation.

There are three main approaches: Program-spectra based (PSB), Model-based diagnosis (MDB) and Program slicing.

Barinel, solution that integrates both PSB and MDB, is, until now, to our knowledge, the option that guarantees the best results. Despite this, the candidates (faulty components) set order does not take into account the real quality of the given component.

With this thesis we want to fix this issue and contribute for a better candidates ordered set, classifying the quality and reliability of each component, using Machine Learning techniques such as Random Forest with language-agnostic information extracted from Git.

Two approaches to the Barinel integration were tried, priors replacement and results modification, and better results were achieved in both.

Keyworks: *Software-fault Localization*, Software Repository Mining, *Machine Learning*, *Classification*

Classification: *Software and its engineering - Software creation and management - Software verification and validation; Computing methodologies - Machine Learning - Machine Learning Approaches*

Resumo

Dada a crescente necessidade de identificar a localização dos erros no código fonte de software, de forma a facilitar o trabalho dos programadores e a acelerar o processo de desenvolvimento, muitos avanços têm sido feitos na sua automação.

Existem três abordagens principais: *Program-spectra based* (PSB), *Model-based diagnosis* (MDB) e *Program slicing*.

Barinel, solução que integra tanto o PSB como o MDB, é, até hoje, com base na investigação feita, a que apresenta melhores resultados. Contudo, a ordenação de conjuntos de candidatos (componentes faltosos) não tem em conta a verdadeira qualidade do componente em causa, mas sim o conjunto de valores que maximizam a probabilidade do conjunto (*Maximum Likehood Estimation* - MLE), devido à dificuldade da sua determinação.

Com esta tese pretende-se colmatar esta falha e contribuir para uma melhor ordenação dos conjuntos, classificando, com recurso a técnicas de Machine Learning como *Naive Bayes*, *Support Vector Machines* (SVM) ou *Random Forests*, a qualidade e fiabilidade de cada componente, através das informações, agnóticas à linguagem de programação usada, disponíveis no sistema de controlo de versões (Software Repository Mining), neste caso *Git*.

Foram experimentadas das abordagens diferentes à integração com o Barinel, subsituição de *priors* e modificação de resultados, e ambas resultaram numa melhor ordenação de resultados.

Palavras-chave: *Software-fault Localization*, Software Repository Mining, *Machine Learning*, *Classification*

Classificação: *Software and its engineering - Software creation and management - Software verification and validation; Computing methodologies - Machine Learning - Machine Learning Approaches*

"Software is eating the world."

Marc Andreessen

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

MSR	Mining Software Repositories
SFL	Spectrum-based Fault Localization
PSB	Program-spectra based
MDB	Model-based diagnosis
MLE	Maximum Likehood Estimation
SVM	Support Vector Machines
MHS	Minimal Hitting Set
SMOTE	Synthetic Minority Over-sampling Technique

Chapter 1

Introduction

With this chapter we aim to frame the subject, explain our motivation and describe the document structure.

1.1 Context

With the big growth of the software development industry, the existence of tools that help programmers do a better job becomes even more important.

It is estimated that the economy of the United States of America loses 60 billion dollars every year in costs associated with the development and distribution of fixes to defects on software and in its re-installation [ZC09]. Given this, we can claim that tools of software fault localization, by helping reduce the time invested in this task, may have a huge positive impact in the economy. In the domain of software fault localization the research is considerable. *Ochiai*, *Tarantula*, *Bayes-A* and Barinel are just some of the existing solutions, being the Barinel algorithm the one that reveals better results. Even though, this algorithm can be optimized, returning more accurate results, by using defect probabilities estimated according to information related to the project.

Given that version control systems, such as *Git*, contain all the project's history and information about all changes (e.g. changed content, date and authors), we believe this data, along with Machine Learning algorithms, may be the key to Barinel, improvement.

1.2 Motivation and Objectives

Given the possibilities of Machine Learning and information extraction from software repositories, with this dissertation we intend to:

- Be able to predict the defect probability of each component in a given software project that uses Git with an useful precision, regardless of the programming language used.

- Optimize the order of results of Barinel.

1.3 Structure

In addition to this chapter, this dissertation contains 6 more chapters.

Chapter 2 describes the state of the art and presents work related to fault localization software, software repository mining and some other approaches to defect prediction. Chapter **??** presents the concept and working details of the solution created to estimate the defect probability. Chapter 4 explains both integrations tried, results modification and priors replacement. Chapters 5 and 6 contain the experimental results obtained by experimentation and the discussion of those results, respectively. Chapter **??** concludes the dissertation by analyzing the goals satisfaction, main contributions and further work.

Chapter 2

State of the art

This chapter includes a literature review and describes state of the art fault localization software, such as Barinel, the different approaches to defect prediction, and Software Repository Mining.

2.1 Fault Localization Software

Fault Localization Software helps to automatically locate code that generates faults when it is executed, reducing the costs associated with finding it, which would have to be done manually by the developer. There are three main types of approach: *Program Slicing*, *Spectrum-based diagnosis* and *Model-based diagnosis*. The *Spectrum-based diagnosis* approaches will have a more detailed analysis due to their greater relevance to this work.

2.1.1 *Program Slicing*

Introduced by Mark Weiser in 1981 [Wei81, Wei82], this method starts its analysis from the fault and, by way of the program's data and control flow, tries to find where the problem originates.

By removing all statements that won't affect the data set or program flow, this method determines a section of code (slice) corresponding to all statements that may be the root of the problem and must be inspected. [PAW04]. By reducing the amount of statements that require inspection, the time needed to fix an error is also reduced.

Usually, a static analysis of the program will result in a huge section of code, so there are other, dynamic, methods that will considerably reduce its size, such as *program dice* or *dynamic program slicing* [PAW04].

2.1.2 *Spectrum-based diagnosis*

Spectrum-based Fault Localization (SFL) is a probability-based fault detection method that will estimate each software component's probability of containing faults by analyzing execution-related

information, whether that execution is successful or not [AZV07]. This method is able to generate good results when the project contains a big amount of test cases and can be executed swiftly, being able to scale well to big projects [MS08].

This method generates a matrix based of data saved during the execution (*program spectrum* [RBDL97]), connecting each test case ran with the components executed and the result, successful or unsuccessful, of that test.

	obs			
	c_1	c_2	c_3	e
t_1	1	1	0	1
t_2	0	1	1	1
t_3	1	0	0	1
t_4	1	0	1	0

Table 2.1: *Hit-spectra matrix*

With this matrix, also called *hit-spectra matrix*, a *similarity coefficient* is calculated for each of the components [AZG09], showing how likely it is that a component may have a fault. The way this coefficient is calculated is different, depending on the algorithm used. For example, *Pinpoint* [CKF+02], *Tarantula* [JH05] and *Ochiai* [AZV07], each, respectively, calculate the coefficient as follows

$$s_J(j) = \frac{a_{11}(j)}{a_{11}(j) + a_{01}(j) + a_{10}(j)} \tag{2.1}$$

$$s_T(j) = \frac{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)}}{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)} + \frac{a_{10}(j)}{a_{10}(j)+a_{00}(j)}} \tag{2.2}$$

$$s_O(j) = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) * (a_{11}(j) + a_{10}(j))}} \tag{2.3}$$

2.1.3 *Model-based diagnosis*

The basis of *Model-based diagnosis* is comparing the model, i.e. the description of the system's behavior, with the actually observed behavior [MS08]. The difference between the two is then used to identify components that may explain errors occurred. This actually requires a formal description of the system, which makes the task harder [PAW04].

In order to facilitate using this method, the model is usually inferred by using the software itself, specifically, by using the test cases defined in it [PAW04].

Although this method generates high-reliability results, the computational effort required to create a model for a big program will prevent it to be used in real projects most of the times [MS08].

2.1.4 Barinel

Barinel is an algorithm inspired by the two methods previously described, *program-spectra based* and *model-based diagnosis*, thus being able to generate better results than the other approaches, albeit with a slightly higher cost [AZG09].

This algorithm starts by analyzing a *hit-spectra matrix*, representing a connection between the executed tests, the executed components, and their final result.

	obs			
	c_1	c_2	c_3	e
t_1	1	1	0	1
t_2	0	1	1	1
t_3	1	0	0	1
t_4	1	0	1	0

Table 2.2: *Hit-spectra matrix*

In the table 2.2, we have identified 3 distinct components (c_1, c_2 e c_3), 4 executed tests (t_1, t_2, t_3 e t_4) and the result of their execution (e). A value of 1 in any of the observation columns (obs) shows that the specific component has been executed in that test, while a value of 0 shows the opposite, the component has not been executed in that test. For the column e, a value of 1 indicates that the corresponding test has failed. For example, the test t_4 has executed the components c_1 and c_3 and completed successfully.

2.1.4.1 Generating candidates

Using this matrix, a list of candidate sets (d) is generated. This list is reduced to the lowest possible amount of candidates, as they encompass all other candidates and allow a reduction of the search space. This problem, entitled *minimal hitting set* (MHS), is itself an *NP-hard* problem, which means specific heuristics had to be created [Rui09, CA13]. The algorithm used by Barinel to resolve this problem is *Staccato*.

In this case, only two candidates would be generated:

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

2.1.4.2 Candidate sorting

For each candidate d, a probability is calculated using a *Naïve Bayes* model:

$$\Pr(d \mid obs, e) = \Pr(d) \cdot \prod_i \frac{\Pr(obs_i, e_i \mid d)}{\Pr(obs_i)} \tag{2.4}$$

$Pr(obs_i)$ is just a normalized term, identical for every candidate and not used for sorting.

Being p_j the *a priori* probability of component c_j causing a fault, also called *prior*, we can define $Pr(d)$, the probability of the candidate being responsible for the error, without taking into account additional evidence, such as

$$\Pr(d) = \prod_{j \in d} p_j \cdot \prod_{j \notin d} (1 - p_j) \tag{2.5}$$

Being g_j (*component goodness*) the probability of component c_j executing correctly if c_j is part of the faulty components set, we have

$$\Pr(obs_i, e_i \mid d) = \begin{cases} \prod\limits_{j \in (d \cap obs_i)} g_j & \text{if} e_i = 0 \\ 1 - \prod\limits_{j \in (d \cap obs_i)} g_j & \text{otherwise} \end{cases} \tag{2.6}$$

Taking our example into account

$$\Pr(d_1 \mid obs, e) = \overbrace{\left(\frac{1}{1000} \cdot \frac{1}{1000} \cdot \left(1 - \frac{1}{1000}\right)\right)}^{\Pr(d)} \times \overbrace{\underbrace{(1 - g_1 \cdot g_2)}_{t_1} \times \underbrace{(1 - g_2)}_{t_2} \times \underbrace{(1 - g_1)}_{t_3} \times \underbrace{g_1}_{t_4}}^{\Pr(obs,e|d)} \tag{2.7}$$

$$\Pr(d_2 \mid obs, e) = \overbrace{\left(\frac{1}{1000} \cdot \frac{1}{1000} \cdot \left(1 - \frac{1}{1000}\right)\right)}^{\Pr(d)} \times \overbrace{\underbrace{(1 - g_1)}_{t_1} \times \underbrace{(1 - g_3)}_{t_2} \times \underbrace{(1 - g_1)}_{t_3} \times \underbrace{g_1 \cdot g_3}_{t_4}}^{\Pr(obs,e|d)} \tag{2.8}$$

In cases where there are unknown g_j values, the value of $Pr(obs, e|d)$ is maximized using the *Maximum Likelihood Estimation* (MLE) algorithm.

In this case, all of g_j values are unknown. Applying the MLE algorithm to both functions and calculating the final result gives us:

- $Pr(d_1, obs, e) = 1.9 \times 10^{-9}$ ($g_1 = 0.47$ e $g_2 = 0.19$)
- $Pr(d_2, obs, e) = 4.0 \times 10^{-10}$ ($g_1 = 0.41$ e $g_3 = 0.50$)

2.1.4.3 *Crowbar*

Crowbar[1], formerly known as GZoltar[2], is the tool used to materialize the Barinel algorithm and that allows the analysis of *Java* projects [CRPA12].

By using code injection and running *JUnit* tests, *Crowbar* is able to identify executed components and connect them to unsuccessful tests.

[1] http://crowbar.io
[2] http://www.gzoltar.com/

There are different ways to present the results, the main one being the *Sunburst* visualization, shown in Figure 2.1, where each ring represents a different granularity level, from the project to a line of code [GCA13].

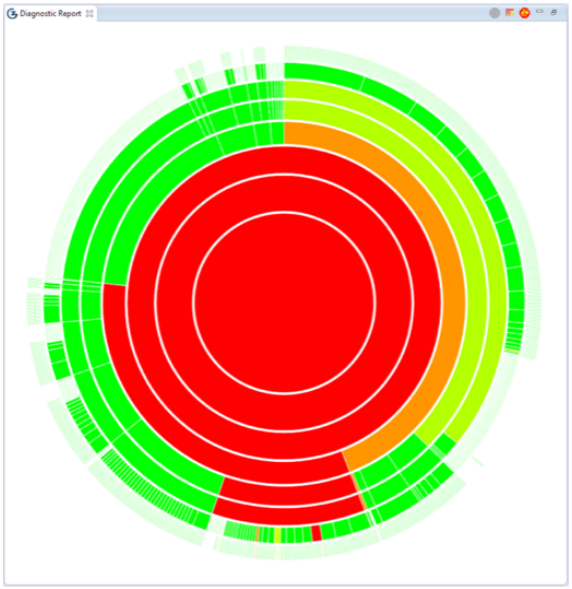


Figure 2.1: Sunburst visualization on Crowbar

In addition to that information, this tool also shows us the component list sorted by fault likelihood.

2.2 Software Repository Mining

Software Repository Mining encompasses all software which is able to extract data from repositories, such as *Git* or *SVN*.

2.2.1 Corrections and defects identification

While extracting information from code repositories, such as *Git*, it is important to identify the types of changes that were made. A text description associated with the change is of the utmost importance for this identification, allowing it to be made with a high degree of precision, depending on the project [MV00].

It is estimated that between 34 and 46% of all changes are corrections and that they represent between 18 and 27% of all added and removed lines of code [MV00]. There were also some patterns found relating code corrections with their size and with the day of the week in which they were executed [ŚZZ05].

After this identification, it is possible to find out which changes have introduced the errors by using the *SZZ* algorithm [ŚZZ05]. This algorithm analyzes the code and uses its history to

correctly locate which *commit* has introduced the defect, ignoring changes that only inserted comments, for example.

2.2.2 Tools

There are several tools to facilitate this process. Let us highlight two of them.

2.2.2.1 *libgit2*

Libgit2[3] is a multi-platform library, without dependences, allowing a direct and native performance interaction with the *Git* repository.

This library can be used with any language supporting *C* bindings, so there are dozens of implementations using languages like *Python*[4], *Javascript*[5] and *Java*[6].

2.2.2.2 *GHTorrent*

GHTorrent[7] is a project with the objective of creating a scalable and easy to query (even *offline*) database, which replicates the information obtained through the GitHub[8] REST API.

This tool works in a distributed way and monitors the GitHub API through the page `https://api.github.com/events`. Each event triggers an information and content extraction to a *MongoDB* database and a different *MySQL* database [GS12].

2.3 Approaches to defect prediction

We have already explored some approaches to code quality prediction. In this chapter we will highlight some methods, like *BugCache*, *FixCache* and *Buggy Change Classification*, which are included in the type of approach we want to follow (*change log*) and that allow us, together with other methods, to optimize defect localization.

2.3.1 *BugCache/FixCache*

Based on the assumption that faults never occur isolated, and therefore where there's a defect others can also exist, *BugCache* creates a list of components which have a high probability of containing defects, based on an analysis of the complete changes history of the project.

This algorithm stands out due to its precision, managing a 73 to 95% accuracy when used with file-level granularity, being the best to date [**?**].

Defects are identified by order of occurrence and added to the list. When that list reaches its maximum size, components start to be removed according to the chosen substitution method.

[3] `https://libgit2.github.com/`
[4] pygit2 - `http://www.pygit2.org/`
[5] nodegit - `https://github.com/nodegit/nodegit`
[6] Jagged - `https://github.com/ethomson/jagged`
[7] `http://ghtorrent.org/`
[8] `http://developer.github.com/`

There are several methods using different metrics, such as *Least Recent Used - LRU*, number of recent defects and number of recent changes [**?**].

With this algorithm, we can conclude [**?**]:

- If a defect has been introduced, there is a tendency to soon introduce additional new defects (*temporal locality*)
- If a component was added or changed recently, it has a higher probability of defect (*changed-entity locality*, *new-entity locality*)
- If a component has introduced an error, the components most directly connected to that one will also introduce errors in the future (*spatial locality*)

The difference between *BugCache* and *FixCache* is the time at which each one updates the list. The former updates the list when a defect is introduced, while the latter only updates when the error is fixed. Due to this difference, implementing *FixCache* is easier.

2.3.2 *Buggy Change Classification*

Change Classification has a notably different approach and a very distinct objective too. *Change Classification*, resorting to *Machine Learning* methods and available information about previous errors, is able to predict if a change has introduced a new defect. This prediction has a high accuracy of 78% [Whi08].

In a first phase, all changes up to that moment are either classified as *buggy* or *clean*. After classifying them and extracting data for each one, a model is created by ẗrainingä classification algorithm.

The types of data used are divided in seven groups: complexity metrics, added code, removed code, file and folder names, new code and meta-data [Whi08].

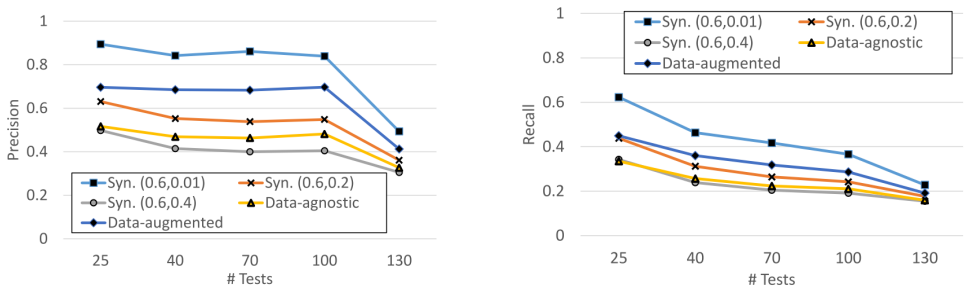
Both *Support Vector Machine* (SVM) and *Naive Bayes* were used in the study, with the SVM-based classifier being the one with better results [Whi08].

2.3.3 Data-Augmented Software Diagnosis

A different approach that also aims to help predict the location of software faults is Data-Augmented Software Diagnosis [ESK15].

Model-based and spectrum-based approaches are able to identify faulty components with high precision and recall, but there is still room for improvement and since most projects use some kind of Version Control Software (VCS), there is a big amount of information that is not used. This opportunity is the foundation of the approach.

Barinel assumes that the defect probability, known as the prior, of every component is uniform, 0.001. However, this approach proved it is possible to optimize Barinel results for Java projects by leveraging information about the project and supervised machine learning algorithms to predict the real component's defect probability.



(a) Precision (b) Recall

Figure 2.2: Diagnosis accuracy as a function of # tests given to the diagnoser. [ESK15]

First, healthy and faulty components are identified across the entire project's history according to Bug Trackers. For each component, a list of features, which is not specified, is extracted. The list is composed of traditional and object-oriented software complexity metrics, such as number of lines of code and cohesion, and values extracted from the software change history, like lines added or removed in last version and age of the file.

With the help of a supervised machine learning algorithm (either Random Forest, J48 or Naive Bayes) a model is created and used to classify each component in the project as healthy or faulty. The confidence that a component is faulty is then used as prior.

Experimental results revealed an increment both on precision and recall, see Figure 2.2. However, the solution was tested just on one project.

Chapter 3

Estimating Defect Probabilities

In order to estimate the defect probability of each software component we developed a data mining application, agnostic to the project's language. It automatically extracts data from the current state and about older defects from a *Git* repository, creates a model and predicts the probability. The project is written in Javascript (Node.js) and Python 3 and heavily uses *node-git* and *scikit-learn*.

In this chapter, we will explain the concept behind it, approach the process used in each of the steps and will explain how to install and use this application.

3.1 Concept

Some patterns are easily recognizable when trying to identify faulty components on a software project and this information could be helpful for software such as Barinel.

We might say, for example that having a high number of recent changes or having been changed by a given junior developer new to the project probably increases the chances of a file having an error, due to past experiences. So, deep down we are just assuming that the knowledge of the meta-data of old faulty components may allow us to predict which components are more probable to be faulty, now.

Based on that assumption, the application, for a given repository state, extracts meta-data knowledge from past commits with faulty components, the parent commits of a fix. For each of these commits, all of components are analyzed and labeled as faulty or clean, if it was or not changed on the fix, respectively.

Since the application should be able to predict the defect probability for any given software project that uses *Git*, static analysis is not used. The only information that is used is information directly related to the file changes, authors, number of lines or bytes.

Along with the extraction of the meta-data information from the past faulty states, it must also extract the meta-data of the given repository state. The data from the first extraction must be used

to create a machine learning model, that should be able to predict the defect probability of each component presents in the current state of the project.

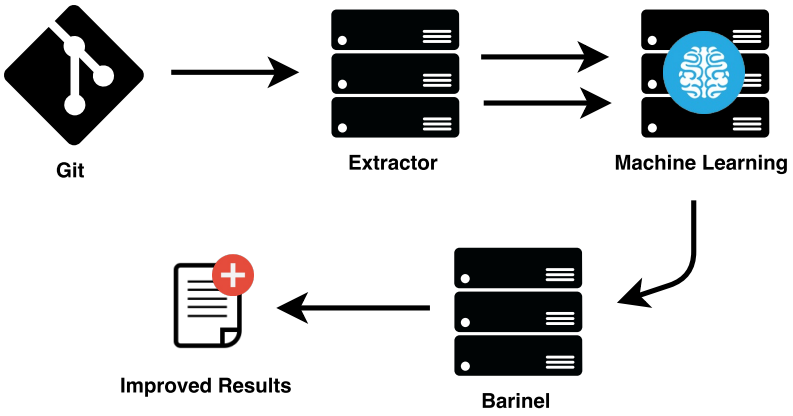


Figure 3.1: Representation of data flow

This concept differs from Data-Augmented Software Diagnosis (Subsection) by being language agnostic, requiring a different set of features mostly focused on number of changes, categorized by type and authors, and weighted according to size and date. It also differs by not requiring that the project uses a bug tracking software and also, as explained in the following section.

3.2 Install

The application relies on *Redis*, *Git*, *Node.js* and *Python 3* and requires some dependencies such as *SciPy*, *scikit-learn*, *numpy*.

```
1  $ git clone --depth 1 https://github.com/atduarte/master-thesis.git
2  $ cd master-thesis/app
3  $ npm install --prod
```

3.3 Usage

Having all the required dependencies installed, the usage is very straight-forward. In order to automatically execute all the steps (extraction, data preparation, modeling and prediction) the following command should be used by replacing "`{project-name}`" and "`{repository-path}`" respectively with the chosen project name and the path to the folder containing the repository we want to analyze:

```
1  $ node index.js all {project-name} {repository-path}
```

It is possible to specify a different classification label (e.g. "`--classification-label=_mostChanged`", default is "`_mostChanged25`"), a different number of estimators used when modeling (e.g. "`--estimators=100`", default is 5) or even to define the level of logging by appending, for example, "`--log-level=verbose`". The existing levels available, from higher to lower priority, are "`error`", "`warn`", "`info`", "`verbose`" and "`silly`". The default log level is "`info`".

```
$  node index.js all math ../../tests/Math001
info extract/extract Raw Extraction started
info extract/extract 499 fix commits found
info prepareJson/prepare JSON Preparation started
info prepareJson/prepare Will prepare 639 files
info results/prepare CSV Preparation started
info results/prepare Got 552 files
info results/prepare Got 203 columns
info results/ml Modeling
```

Some extra configurations are also available by creating a project configuration file, in the folder "`project-config/`", with the same name as the `project-name` plus ".js". It allows to change the *regex* used to identify the fix commits, to define a file filter and to define an email normalizer, in order to improve the quality of the extracted data .

```
'use strict';

module.exports = {
    fileFilter: (filename) => {
        filename = filename.toLowerCase();

        return filename.endsWith('.java') // Is Java
            && !filename.startsWith('src/test'); // Aren't tests
    },
};
```

Before terminating, the application will create a file ("`prediction.e500._mostChanged25.csv`", by default) in the repository folder containing the predicted defect probability for each source code file.

```
...
src/main/java/org/joda/time/Chronology.java,0.0
src/main/java/org/joda/time/DateMidnight.java,0.21
src/main/java/org/joda/time/DateTime.java,0.484
src/main/java/org/joda/time/DateTimeComparator.java,0.0
src/main/java/org/joda/time/DateTimeConstants.java,0.0
src/main/java/org/joda/time/DateTimeField.java,0.006
```

```
src/main/java/org/joda/time/DateTimeFieldType.java,0.002
src/main/java/org/joda/time/DateTimeUtils.java,0.966
src/main/java/org/joda/time/DateTimeZone.java,0.414
src/main/java/org/joda/time/Days.java,0.0
...
```

In order to be able to execute only some step of the process, there is the possibility of executing only one operation. The available commands are:

- "`raw`" - Extracts data from *Git* and saves it to "`out/{project-name}/raw`"
- "`json`" - Processes the raw data and converts it to a new JSON structure. Results are saved to "`out/{project-name}/json`"
- "`results`" - Creates the CSV files for training and prediction data, based on the JSON files, creates the model and predicts the defect probability. Final result is saved at "`{repository-path}/prediction.csv`".

3.4 Process

As stated before the process is divided in extraction, data preparation, modeling and prediction steps. We will now dive deep into each.

3.4.1 Extraction

The first objective is to get the list of commits to analyze: the HEAD commit and all the fix commits that preceded it. So, first, the *HEAD* commit is identified. The application then walks recursively through each parent commit or commits and, if it is message matches the following regex and it is not a merge, adds it to the list of commits to analyze.

```
(\b|)fix(|\b|ed|ing)|bug( | \#|\-|)[0-9]+
```

Having this list, it analyzes each commit individually, ignoring the ones that were already extracted or that have no changed files after filtering according to configuration. First the information directly related to the commit and its tree is extracted:

- id - Id
- message - Message
- date - Date
- author - Author
- components - List of components

Then, the application walks through the changes history of each component, following name changes and extracting this info for each commit:

- id - Id
- date - Date
- author - Author
- parentCount - Parent count
- isFix - Is it a fix?
- filename - Filename
- lines - Number of lines
- byteSize - Size of file in bytes
- linesAdded - Number of lines added
- linesRemoved - Number of lines removed

This data is then saved to a file named according to the commit id, in the "`raw`" folder (e.g "`out/math/raw/17d6f2163db436518f953166c1e9d495232f90b6`").

```
{
  "id": "0b1b9a9dc86da871ce5e7839b1b2df13c99dd9f8",
  "message": "Submitted Javadoc fixes from Andreou Andreas ...",
  "date": 1055343030,
  "author": "tobrien@apache.org",
  "components": {
    "src/java/org/apache/commons/math/ContractableDoubleArray.java": {
      "linesAdded": 3,
      "linesRemoved": 3,
      "changes": [
        {
          "id": "8b62ed457040b6a1b4562aa0d0df88e1e77bddce",
          "date": 1053499586,
          "author": "tobrien@apache.org",
          "parentCount": 1,
          "isFix": false,
          "filename": "src/java/org/apache/commons/math/ContractableDoubleArray.
              java",
          "lines": 322,
          "byteSize": 12970,
          "linesAdded": 54,
          "linesRemoved": 3
        },
```

```
        ...
      ]
    }
  }
}
```

Since this procedure has a high computation cost, some enhancements have been made. Considering that the modeling step will balance the training data set, reducing the number of *clean* components according to the number of existing *faulty* components, the system randomizes the list and limits the extraction of *clean* components up to a maximum of four times the number of *faulty* components on the given commit:

```
const _ = require('lodash');

if (notHead) {
  const changedComponents = _.pickBy(info.components, x => x.linesAdded + x.
      linesRemoved > 0);

  // ...

  const cleanComponentNames = _(info.components)
      .pickBy(x => x.linesAdded + x.linesRemoved == 0)
      .keys()
      .shuffle()
      .splice(0, 4 * _.size(changedComponents)).value();

  info.components = Object.assign({}, changedComponents,
      _.pick(info.components, cleanComponentNames)
  );
}
```

It also relies heavily on caching. Since in most cases the commits analyze common files, we can cache some information and improve performance. Due to the ability to merge two different commits, *Git* can have a non-linear history. As a result iterating recursively through all parent commits for each analyzed commit is necessary in order to safely determine which parent commits changed which files. Although, for any given file and commit, this list will most of the times have common elements with lists from changes of the same component for other commits. Given this, the data that is extracted for each item on the list is cached in *Redis*, using file name and commit as reference.

Figure 3.2 illustrates a possible Git history. We will assume that only one file exists, F, it was changed in all commits and will also assume commit $C1$ and commit $C2$ as fix commits. The application would extract data for $HEAD$, $C4$ and $C5$ and for F each time. When extracting F on $HEAD$ the list of changes would be $[C6, C5, C4, C3, C2, C1]$, on $C5$ it would be $[C3, C2, C1]$ and on $C4$ it would be $[C2, C1]$. Then for each change data, such as number of lines added and

lines removed, have to be extracted. As we saw, extraction happens for each analyzed commit, for each file, for each past file change and takes some time. Without cache, it would extract $[C6, C5, C4, C3, C2, C1, C3, C2, C1, C2, C1]$, but since the results are cached it just runs for $[C6, C5, C4, C3, C2, C1]$, improving performance significantly.

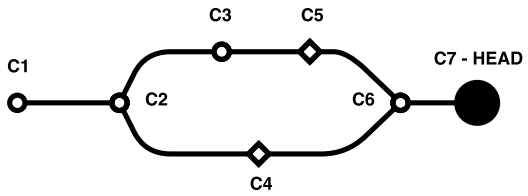


Figure 3.2: Example representation of commits history, with fix commits illustrated in a diamond shape

3.4.2 Data Preparation

This step is the simplest. The only task is to convert the raw JSON structure, focused on the commit, to a JSON structure, focused on the component. 7 Iterating through each file at the "`raw`" folder and creating a new one with the same name in the "`json`" folder.

The data contained in this new file will be ready for being converted to CSV and to be used for modeling or prediction.

The following columns are prepared for each component:

- __changed - Was it changed?
- __filename - Component name
- _lines - Number of lines
- _bytes - File size in bytes
- _mostChanged - Was it the most changed file in this commit?
- _mostChanged25 - Was this file in top 25% of most changed files is this commit?
- _mostChanged50 - Was this file in top 50% of most changed files is this commit?
- _mostChanged75 - Was this file in top 75% of most changed files is this commit?
- changes - Number of changes made in the past
- changes:date-weighted - Sum of the result of the time-weighted function applied to each change
- changes:size-weighted - Sum of the number of lines added and removed in each change

- changes:date+size-weighted - Sum of the result of the time-weighted function applied to each change multiplied for the number of lines added and removed in it
- authors - Number of authors that changed this file
- authorChanges::author - Number of changes made in the past by a specific author. E.g. authorChanges::luc@apache.org
- authorChanges:date-weighted:author - Sum of the result of the time-weighted function applied to each change made by a specific author. E.g. authorChanges:date-weighted::luc@apache.org
- authorChanges:size-weighted:author - Sum of the number of lines added and removed in each change made by a specific author. E.g. authorChanges:size-weighted::luc@apache.org
- authorChanges:date+size-weighted:author - Sum of the result of the time-weighted function applied to each change made by a specific author multiplied for the number of lines added and removed in it. E.g. authorChanges:date+size-weighted::luc@apache.org

Being t the normalized timestamp of the change, where 0 is the timestamp of the first commit and 1 of the latest, and $W = 0.5$, the time-weighted function used was the following:

$$\frac{1}{1+e^{(-12\cdot t)+(1-W)\cdot 10+2}} \tag{3.1}$$

In order to improve modeling results, we also created the columns "changes-others", "changes-others:date-weighted", "changes-others:size-weighted", "changes-others:date+size-weighted", "changes-fixes", "changes-fixes:date-weighted", "changes-fixes:size-weighted" and "changes-fixes:date+size-weighted" that are equal to the already existing columns starting with "changes", but refer only to changes that were not fixes and changes that were fixes, respectively.

Plotting the results we noticed that the date influenced significantly the data related to changes, since older commits have less history behind. So for each, we added a normalized version by the commit.


```
[
  {
    "__changed": true,
    "__date": 1364521362,
    "__filename": "src/main/java/org/apache/commons/math3/distribution/fitting/
        MultivariateNormalMixtureExpectationMaximization.java",
    "_lines": 441,
    "_bytes": 18211,
    "_mostChanged": false,
    "_mostChanged25": true,
    "_mostChanged50": true,
    "_mostChanged75": true,
    "changes:raw": 3,
```

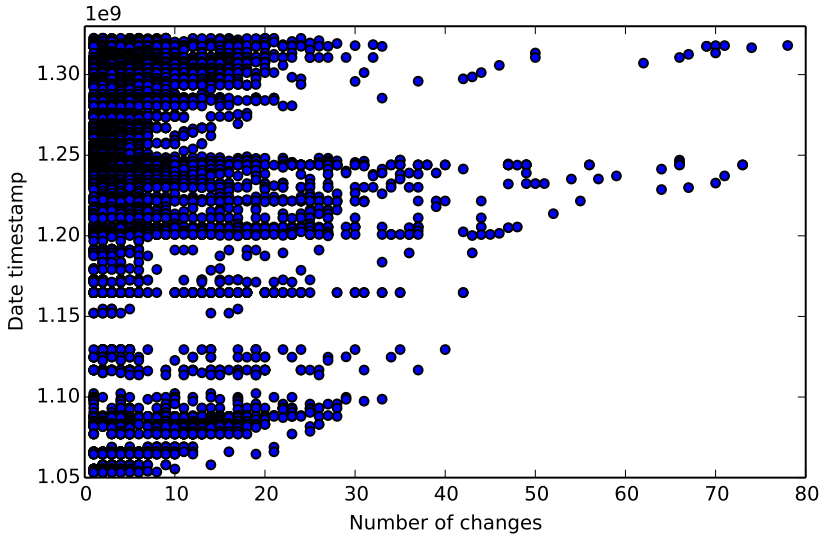


Figure 3.3: Example of the influence of the date on number of changes. (Project Math, Defect 1)


```
    "changes:normalized": 0.2857142857142857,
    "changes-fixes:raw": 0,
    "changes-fixes:normalized": 0,
    "changes-others:raw": 3,
    "changes-others:normalized": 0.3333333333333333,
    "changes:date-weighted:raw": 2.979453,
    "changes:date-weighted:normalized": 0.290809968305134,
    "changes:size-weighted:raw": 481,
    "changes:size-weighted:normalized": 0.4746494066882416,
    "changes:date+size-weighted:raw": 477.701982,
    "changes:date+size-weighted:normalized": 0.4831911672918317,
    "changes-fixes:date-weighted:raw": 0,
    "changes-fixes:date-weighted:normalized": 0,
    "changes-fixes:size-weighted:raw": 0,
    "changes-fixes:size-weighted:normalized": 0,
    "changes-fixes:date+size-weighted:raw": 0,
    "changes-fixes:date+size-weighted:normalized": 0,
    "changes-others:date-weighted:raw": 2.979453,
    "changes-others:date-weighted:normalized": 0.3394176672358086,
    "changes-others:size-weighted:raw": 481,
    "changes-others:size-weighted:normalized": 0.4845814977973568,
    "changes-others:date+size-weighted:raw": 477.701982,
    "changes-others:date+size-weighted:normalized": 0.49340824785460163,
    "authors:raw": 3,
    "authors:normalized": 0.6666666666666666,
    "authorChanges::luc@apache.org:raw": 1,
    "authorChanges::luc@apache.org:normalized": 0,
    ...
    "authorChanges:date-weighted::luc@apache.org:raw": 0.993164,
```

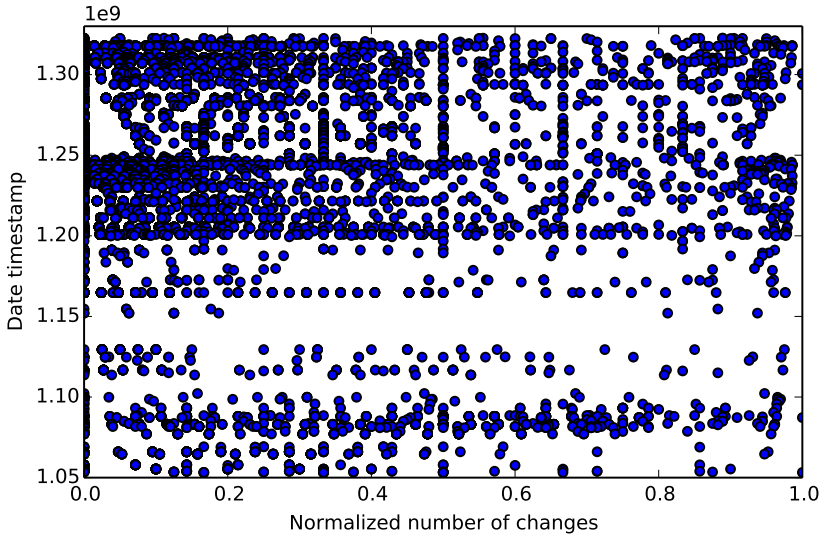


Figure 3.4: Example of the relation between the date on the normalized number of changes. (Project Math, Defect 1)

```
42      "authorChanges:date-weighted::luc@apache.org:normalized":
            0.00025474141229450394,
43      "authorChanges:size-weighted::luc@apache.org:raw": 33,
44      "authorChanges:size-weighted::luc@apache.org:normalized": 0.5454545454545454,
45      "authorChanges:date+size-weighted::luc@apache.org:raw": 32.774417,
46      "authorChanges:date+size-weighted::luc@apache.org:normalized":
            0.5455704368440787,
47      ...
48    },
49    ...
50  ]
```

The data from the multiple fix commits is then combined in a file named "history.csv" and the data from the *HEAD* commit is placed at "master.csv". These two files will be only files required to execute the next step.

3.4.3 Modeling and Prediction

For modeling and predicting *Python 3* and *scikit-learn* is used. Random Forests, Decision Trees and Naive Bayes were tried, but the former achieved significantly better results as it is presented in the Experimental Results chapter 5. Neural Networks were not used since the training set is normally small.

Before training the model, the training set is randomized and balanced, first to a ratio of 5 *clean* component for each *buggy* component and then SMOTE is used to create more *buggy* component entries. In order to train the model a *Stratified KFold* (4 folds) is used and run 10 times, generating

40 different models. Performance of these 40 models is measured according to the difference of the mean defect probability of buggy test data and clean test data. The 15 models with the highest difference are then used to predict the defect probability of each component in the current project state and a mean is calculated.

As referred before, the results are saved directly at the repository folder.

Chapter 4

Barinel Integration

One of the objetives is to improve Barinel results and for doing so integrations with the existing Barinel project were made. Priors replacement and results modification were the approaches chosen.

4.1 Priors Replacement

Barinel by default uses $\frac{1}{1000}$ as the defect probability (prior) for all software components (4.1).

$$\Pr(d) = \prod_{j \in d} \frac{1}{1000} \cdot \prod_{j \notin d} (1 - \frac{1}{1000}) \tag{4.1}$$

With this integration the Barinel project is now capable of receiving specific priors for each component, reading it from a CSV file located at the root of project being analyzed, and attributing them to the corresponding probes (4.3).

$$\Pr(d) = \prod_{j \in d} p_j \cdot \prod_{j \notin d} (1 - p_j) \tag{4.2}$$

Maximum Likehood Estimation (MLE) continues to be used in order to maximize the probability, by defining the best possible goodness values.

4.2 Results Modification

The second approach maintains the priors at $\frac{1}{1000}$ and directly modifies the final result calculated by Barinel ($Pr(d_j, obs, e)$) by multiplying it by 2 if considered faulty.

$$Pr'(d_j, obs, e) = Pr(d_j, obs, e) \cdot \begin{cases} 2 & \text{if } j is faulty \\ 1 & \text{otherwise} \end{cases} \tag{4.3}$$

Chapter 5

Experimental Results

In order to reliably validate the defect probability prediction we used an existing database of faults, *Defects4J*. This database contains a list of commits with defects and their location from five different projects (JFreechart, Closure compiler, Apache commons-lang, Apache commons-math and Joda-Time). Since compatibility with Barinel and using *Git* is required, only the last three projects were used, making a total of 184 commits.

Project name	**Shortname**	**Defects**
Apache commons-lang	Lang	59
Apache commons-math	Math	101
Joda-Time	Time	25

Table 5.1: Test Set

5.1 Estimating Defect Probability

First we must know the characteristics of each project in order to better interpret the results, so a table was made containing information about the first and last commit of each project.

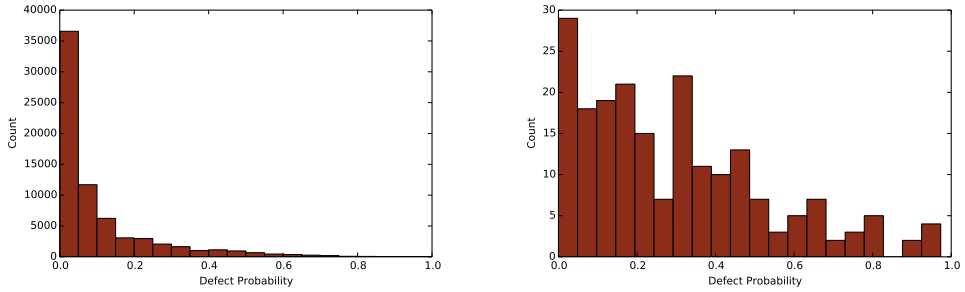
Project	**Defect**	**Files**	**Previous commits**	**Previous fix commits**	**Contributors**
Lang	1	108	3569	366	40
Lang	59	81	1533	190	25
Math	1	813	4877	564	28
Math	103	370	957	66	15
Time	1	162	1717	234	27
Time	26	733	1474	195	9

Table 5.2: Examples of project states from the test set

Three different classification algorithms were tested: Support Vector Machines, Random Forests and AdaBoost. Our preliminary tests showed that Random Forests is the algorithm that provides

an higher accuracy, that using the normalized values did not help and the best label to use for training is `_mostChanged`, so we will focus on this test case.

Figures 5.1 illustrates the different defect probabilities distributions of faulty and probably clean components, across all the 184 defects, respectively.



(a) Clean (b) Faulty

Figure 5.1: Distribution of predicted defect probabilities

Since we used Stratified KFold we are able to determine the prediction accuracy of the model on the train data. Figure 5.2 illustrates the distribution of the mean accuracy of the 15 models used for evaluate each state of the projects.

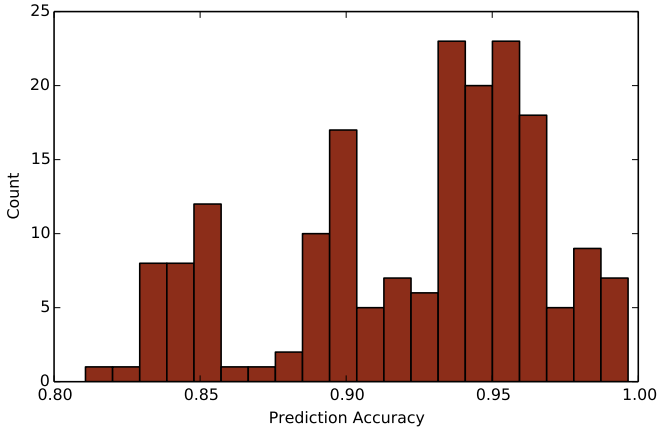
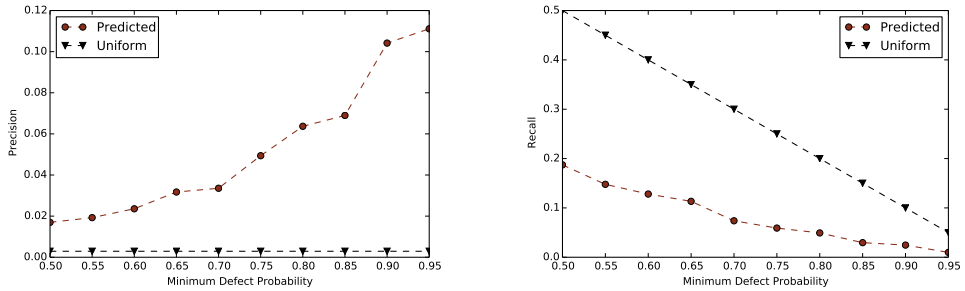


Figure 5.2: Histogram of the mean accuracy of the models used for evaluating each project state at predicting KFold's test data

Precision and recall is compared between our solution and an uniform defect probability distribution for clean and faulty components is exhibited at Figure 5.3.

However since we want the faulty component to be the one with the highest defect probability, it is important not only to verify the defect probability for the faulty component, but also to compare it to the probabilities of the other, probably clean, components in the project. Figure 5.4 illustrates a case where the faulty component has a low defect probability, 12%, but it still is on the top 17%.



(a) Precision

(b) Recall

Figure 5.3: Prediction accuracy of identifying faulty components as a function of the minimum defect probability to consider as faulty, comparing our approach to an uniform defect probability distribution for clean and faulty components

Figure 5.5 exhibits the relative position of the faulty component for all the 184 defects and allowed to concluded that the average percentage of components with higher defect probability than the faulty component is just 16.6% and the median is 10.6%.

5.2 Barinel Integration

Two types of integration with Barinel were tested so the results will be presented separately in the following two sub-chapters. However, in the interest of better understanding the results, an analysis of the results of the unmodified Barinel was made.

The unmodified Barinel analysis, represented on figure 5.6, showed that on 44.57% of the 184 project states the faulty component already are on the first position and on 5.98% it has associated a 0% probability of being faulty. So, no improvements can be made on 50.55% of the examples of the test set. It also showed that in 88.04% of the results the component with the defect is above the 10th position. For the sake of this analysis, in case of draw, we consider the best position and will not consider in Figure 5.6 the cases where the associated probability is 0%.

5.2.1 Results Modification

First, to be able to contextualize the results, the best and worst scenarios were tested. If the predicted defect faulty was 1 to all the faulty components and 0 for all the others, there would be an improvement on 37 cases. If, for instance, the probabilities were totally wrong, it would worsen 54 cases.

Given the predicted defect probabilities is important to determine the best value to use as the minimum for a component to be considered possibly faulty. When considered possibly faulty, as explained in 4, the Barinel fault probability is doubled. For each value from 0.5 to 1, at 0.05 steps, the gain, loss and delta was calculated, as we can see in 5.7.

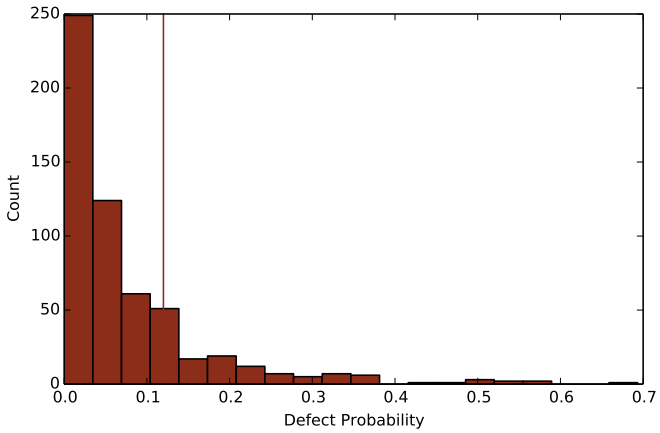


Figure 5.4: Histogram of predicted defect probabilities for Project Math (Defect 50), with a vertical line identifying the faulty component prediction

5.2.2 Priors Replacement

Experimental results showed that some tests on the Math project are flaky[1], making the Barinel results non-deterministic. Given this, only the defect from the Time and Lang projects will be considered and compared.

Since Barinel computation is computationally heavy, just one minimum defect probability (0.65) was tested. In this case, all software components that were classified with an equal or higher defect probability than 0.65 will have a prior of 0.002. All the others will remain with 0.001.

The best case scenario was computed and showed that even with perfect precision it can worsen results. 14 improved results (9 from Lang, 5 from Time) and 3, all from Lang, worsened.

Using the prediction output to change priors resulted in 6 improved results (5 from Lang, 1 from Time) and no worsened test.

[1] A flaky test is a test which could fail or pass for the same configuration

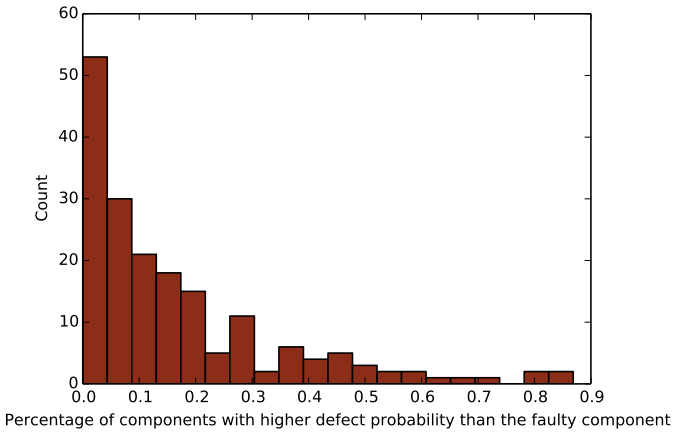


Figure 5.5: Histogram of the percentage of components with higher defect probability than the faulty component for all 184 defects

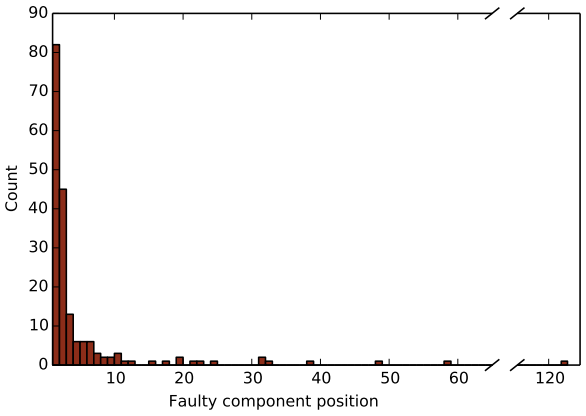


Figure 5.6: Histogram of the relative position of all faulty components, with a probability above zero, reported by the unmodified Barinel

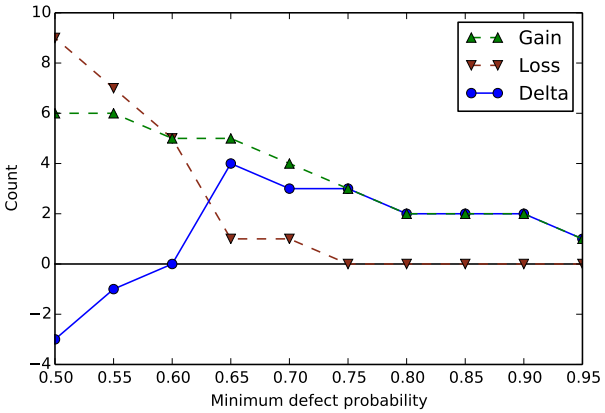


Figure 5.7: Results modification effect on Barinel results, by minimum defect probability

Chapter 6

Discussion

The experimental results acknowledged that extracting the component changes meta-data is valuable by allowing to predict, with good precision and independently of the language, which files are more probable to have defects (Figure 5.5).

6.1 Estimating Defect Probability

In figure 5.1 we clearly see two distinct types of distributions. Probably clean components predicted defect probability tends to 0, as intended, but the same can not be interpreted for the buggy components, whereas the distribution is more uniform. Preferably, the buggy components predicted defect probability distribution would tend more expressively to 1.

Precision and recall of the estimation of defect probability exhibited in Figure 5.3 is also relevant to analyze. The precision improvement we see on this figure over an uniform probability distribution for clean and buggy illustrates the information gain we obtain with this solution.

However, the mean accuracy obtained when classifying the test folds selected by Stratified KFold (Figure 5.2) does not seem to be reflected when classifying the test set, namely the project's state. This could be explained by a overfitting mistake, but we think this is not the problem. We analyzed it, tried to use normalized values for changes, instead of the raw value plus date, and tried to use less features and tested the various options by cutting out more recent data and using the rest to model using KFold. The accuracy of classifying the most recent data was always much lower than the accuracy predicting data that was from a closer time frame.

This may be explained by the fact that evolution of the project may affect which patterns allow to identify faulty components, making data from within a closer time frame, more valuable.

The inability to have more consistent results of the mean accuracy, that vary mainly between 0.8 and 0.95 as illustrated in Figure 5.2 may be caused by the data imbalance and noise.

Since in each fix commit just a small percentage is changed and all the others are considered clean, extracted data is extremely unbalanced and the number of fault components in the training set is small. Using SMOTE improved the results, but the tendency to 0 continues to be noticeable.

The assumption that all unmodified files are clean may also introduce noise in the data, by labeling faulty components as clean. This along with the uncertainty of the identification of fix commits by the commit message, that may lead to wrongly labeling clean components as faulty, may blur the difference between the two types and lower the prediction accuracy.

Besides, the faulty component may contain a defect just because of a recent change in a related component and since the data set does not contain any information about the components relations, it would be difficult for it to have a high predicted defect probability.

Yet, even with so much aspects that may affect our prediction accuracy, the figure 5.5 shows that, in fact, the faulty components tends to be classified as one of the components with the highest defect probability. Which is crucial to our goal of improving Barinel results.

6.2 Barinel Integration

Analysis of the unmodified Barinel, illustrated in 5.6, showed how good the results are and the tenuous percentage of tests that can be improved by using our approach to modify Barinel results

6.2.1 Results Modification

In the best case scenario, the results modification integration can improve 14.67% of the tests. While in the worst case scenario, 29% of the tests would worsen.

Figure 5.7 shows that when considering as faulty all the components with a predicted defect probability above 0.6 the Barinel results improve, with little or no error. Examining for example the results for 0.65 of minimum predicted probability, where the delta is higher, 13.5% of the possible improvements occurred and just one test worsened. Increasing the minimum diminishes both the number of improvements and errors, but starting at 0.75 errors are completely eliminated.

6.2.2 Priors Replacement

The best case scenario showed that even with 100% precision the priors replacement integration can result in worsened tests. [TODO] Explain why clearly

Even though, real results revealed to be promising by improving approximately 43% of the possible tests and not damaging any. This may illustrate how important the defect probability prediction based on language agnostic features is to improve Barinel results.

6.3 Threats to Validity

There are some threats to the validity of this research. The first is the fact that the Math project (101 tests of 184) appeared to have flaky tests, since with the exact same configuration Barinel, which is deterministic, reported some value changes.

Using three open source Java projects, with 184 tests, may also not be sufficient to predict the application behavior in other different projects

Being this research all about defect probabilities, we know that the application made to estimate the defect probability can also have defects and may somehow affect the predictions. Although the application was heavily tested and many results were manually checked for validity.

Last but not least, the data mining application is nondeterministic. It creates models every time and chooses the most accurate ones, in order to avoid being able to predict completely different results, but there will be most certainly differences as expected in a data mining project.

Discussion

Chapter 7

Conclusions and Further Work

We consider that building tools to help developers do a better job is crucial and have a huge positive impact on the global economy.

By reviewing the available literature, it is possible to conclude that interest exists in bridging knowledge from the domain of Artificial Intelligence to Software Engineering. This research aims to contribute to it.

7.1 Goals contribution

We set the goal at building a project that would be able to predict the component defect probability of any software project, written in any language, that uses *Git*, with enough precision to improve Barinel results and we achieved it.

Even with experimental results showing that Barinel already classified the faulty component as one of the top 10 probably faulty components in 88.04% of its results.

So we can conclude that the project was successful and achieved all the defined goals.

7.2 Main Contributions

Results confirm that there is sufficient information available at the Git repository to be able to improve 14.67% of the possible Barinel results, with no loss. It was also clear that priors replacement even if optimal can result in worse results.

A different approach to Barinel results optimization was introduced with Results modification, which, with optimal software component defect probability classification, does never negatively affect results.

Data mining application is open-source and can be already used with any *Git* project to help identifying code hot spots. We consider that this capacity can help save time, by for example advising software developers doing code reviews to be more careful about specific files.

7.3 Further Work

Although the development and validation proved its concept and prediction capability, there is still room for development and future enhancements. Following are some interesting ideas of future work.

7.3.1 Improving fix commits identification

In order to not depend on the usage of a bug tracker software, the fix commits identification is made by analyzing the commit message.

Since using this approach may introduce noise in the train set, which may lower the accuracy of the model, the bug tracker software information could be used when existing.

Other possible improvement is to enhance the regex used for the identification.

7.3.2 Adding static code analysis features, when available

The project showed that it is possible to not use static code analysis features to train a defect classifier. Nonetheless using those features would probably improve the model's accuracy.

So, implementing such features for a set of languages and using them when possible could present interesting results.

7.3.3 Improving machine learning model accuracy

As stated in Chapter 6, the train data set can be noisy, unbalanced and seems to evolve different patterns with time, making it difficult to have good accuracy predicting data by training a model with much older information. Given this, newer train data should have a higher weight than older data when training the supervised machine learning model.

There are a wide range of different approaches to these problems and we have believe they could have a positive impact on the model's accuracy.

7.3.4 Integrating with *GitHub*

This project could be used to warn developers directly at *GitHub* [1] about code hotspots.

For example, every time a pull request is created, the data mining application could analyze the project and if any changed file has a high defect probability or any changed files have their defect probability significantly increased a automatic comment would be made, alerting the submitter and the reviewers.

[1] https://github.com

7.3.5 Creating a different data structure to represent commit history tree

Even with the performance improvements made through ignoring repeated tasks and caching the application is rather slow extracting the information from *Git* and can benefit from more enhancements.

Project's performance can be enhanced, for example, by saving the commit history tree in a data structure that allows to more easily find past commits that changed a given file.

7.3.6 Parallelization

The extraction runs on *node.js* so only one core is used at a time. Parallelization would improve application's performance and is possible, for example, using worker processes.

7.3.7 Improve software component labeling

There is some uncertainty associated with the labeling used in this project. It can be improved if data is interpolated between commits.

For example, if in commit A the file X was faulty and in the commit B the file was exactly the same, both should be labeled as faulty.

References

[AZG09] Rui Abreu, P Zoeteweij, and a J C Van Gemund. Spectrum-Based Multiple Fault Localization. *Automated Software Engineering 2009 ASE 09 24th IEEEACM International Conference on*, pages 88–99, 2009.

[AZV07] Rui Abreu, Peter Zoeteweij, and Arjan J C Van Gemund. On the accuracy of spectrum-based fault localization. *Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007*, pages 89–98, 2007.

[CA13] Nuno Cardoso and Rui Abreu. MHS2: A Map-Reduce Heuristic-Driven Minimal Hitting Set Search Algorithm. 2013.

[CKF+02] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, 2002.

[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*, page 378, New York, New York, USA, sep 2012. ACM Press.

[ESK15] Amir Elmishali, Roni Stern, and Meir Kalech. Data-Augmented Software Diagnosis. 1:247–252, 2015.

[GCA13] Carlos Gouveia, José Campos, and Rui Abreu. Using HTML5 visualizations in software fault localization. *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, 2013.

[GS12] Georgios Gousios and D. Spinellis. GHTorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21. IEEE, jun 2012.

[JH05] J.a. James a Jones and Mary Jean M.J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *Automated Software Engineering*, pages 282–292, 2005.

[KZPW06] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E James Whitehead Jr. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.

[MS08] Wolfgang Mayer and Markus Stumptner. Evaluating Models for Model-Based Debugging.pdf. pages 128–137, 2008.

REFERENCES

[MV00] a. Mockus and L.G. Votta. Identifying reasons for software changes using historic databases. *ICSM conf.*, pages 120–130, 2000.

[PAW04] Alexandre Perez, Rui Abreu, and Eric Wong. A Survey on Fault Injection Techniques. 1:171–186, 2004.

[RBDL97] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *ACM SIGSOFT Software Engineering Notes*, 22(6):432–449, 1997.

[Rui09] Arjan J. C. van Gemund Rui Abreu. A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis. 2009.

[ŚZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30(4):1, jul 2005.

[Wei81] Mark Weiser. Program slicing. pages 439–449, mar 1981.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[Whi08] E.J. Whitehead. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, mar 2008.

[ZC09] Michael Zhivich and Robert K. Cunningham. The real cost of software errors. *IEEE Security and Privacy*, 7(2):87–90, 2009.