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

TODO

**Keyworks**: *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*

# 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 disponíveis no sistema de controlo de versões (*Software Repository Mining*), neste caso *Git*, como por exemplo: número de vezes que foi modificado, número de contribuidores, data de última alteração, nome de últimos contribuidores e tamanho das alterações.

A investigação já feita, revelou a existência de algumas soluções de análise preditiva de *software*, como *BugCache*, *FixCache* e *Change Classification*, capazes de identificar componentes com grande probabilidade de falhar e de classificar as revisões (*commits*) como faltosas ou não, mas nenhuma soluciona o problema.

Este trabalho visa também a integração com o *Crowbar* e a contribuição para a sua possível comercialização.

**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

# Introdução

[TODO] Translate...

Pretende-se com este capítulo enquadrar o tema, explicar as motivações e descrever sucintamente a estrutura deste documento.

## 1.1 Contexto/Enquadramento

Com o elevado crescimento da indústria de desenvolvimento de software, torna-se cada vez mais importante a existência de ferramentas que auxiliem os programadores a desenvolvê-lo mais eficientemente.

Estima-se que a economia dos Estados Unidos perca entre 60 mil milhões de dólares por ano em custos associados ao desenvolvimento e distribuição de correções para defeitos de *software* e na sua reinstalação [**?**]. Pelo que, podemos afirmar que as ferramentas de localização das falhas de *software* (*Software Fault Localization*), ajudando a reduzir o tempo investido nesta tarefa, poderão ter um impacto significativo na economia. Nesta área os avanços são consideráveis. *Ochiai*, *Tarantula*, *Bayes-A* e *Barinel* são apenas algumas das soluções existentes, sendo o algoritmo *Barinel* aquele que apresenta melhores resultados. [**?**] Apesar dos bons resultados apresentados pelo *Barinel*, este poderá apresentar resultados ainda mais rigorosos se tivermos informações relativas ao projeto, como a probabilidade média de erro ou a probabilidade de dado componente, que o constitui, ter defeitos.

Ferramentas de controlo de versões, como o *Git*, uma vez que mantêm todo o histórico do projeto e informações relacionadas com as diversas alterações (p.e. conteúdo, data e autores), em conjunto com técnicas de *Machine Learning*, poderão ser a chave para a melhoria deste algoritmo.

## 1.2 Motivação e Objetivos

Tendo em conta as possibilidades que a extração de dados de repositórios de controlo de versões e o *Machine Learning* nos dão, pretende-se com esta dissertação:

- Being able to predict the defect probability of each component in a given software project, that uses Git, with an useful precision.
- Optimize the order of results of Barinel

## 1.3 Estrutura

# Chapter 2

# State of the art

Neste capítulo é feita uma revisão bibliográfica e descrito o estado da arte do *software* de localização de falhas, como o *Barinel*, das diferentes abordagens de predição de defeitos e ainda de *Software Repository Mining*.

## 2.1 *Fault Localization Software*

O *Fault Localization Software* auxilia na localização automática do código que origina falhas na sua execução, diminuindo o custo desta identificação que teria de ser feita manualmente pelo programador. Existem três categorias principais: *Program Slicing*, *Spectrum-based diagnosis* e *Model-based diagnosis*. Sendo que as abordagens *Spectrum-based diagnosis* serão analisadas mais ao detalhe devido à sua maior importância para este trabalho.

### 2.1.1 *Program Slicing*

Introduzida por Mark Weiser em 1981 [**?**, **?**], esta técnica começa a análise a partir da falha e através do fluxo de dados e de controlo do programa tenta chegar à origem do problema.

Removendo todas as operações que não afetam os dados ou o fluxo do programa, este método define uma porção de código (*slice*) que diz respeito às operações que poderão ser a causa do problema e que terão de ser inspecionadas [**?**]. Ao reduzir o número de operações que necessitam de ser analisadas, o tempo necessário para a correção do erro diminui.

Normalmente uma análise estática do programa resulta numa porção de código grande, havendo outras técnicas, dinâmicas, que reduzem consideravelmente o seu tamanho, como *program dice* ou *dynamic program slicing* [**?**].

### 2.1.2 *Spectrum-based diagnosis*

*Spectrum-based Fault Localization* (SFL) é uma técnica estatística de deteção de falhas que calcula a probabilidade de cada componente de *software* conter falhas, através da análise de informação relativa às execuções, bem sucedidas ou falhadas [**?**]. Esta técnica apresenta bons resultados quando o projeto têm um número elevado de casos de teste e é capaz de executar num tempo reduzido, escalando bem para projetos grandes [**?**].

Esta técnica gera uma matriz, com base nos dados guardados durante a execução (*program spectrum* [**?**]), que relaciona as execuções de casos de teste, com os componentes que executou e com o sucesso ou insucesso do mesmo.

| | *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*

Com esta matriz, também denominada *hit-spectra matrix*, é calculado o coeficiente de similaridade (*similarity coefficient*) para cada um dos componentes [**?**], que corresponde à probabilidade desse componente ter uma falha. A forma como este coeficiente é calculado difere de algoritmo para algoritmo. Dando como exemplos o *Pinpoint* [**?**], o *Tarantula* [**?**] e o *Ochiai* [**?**], que têm os coeficientes respetivamente calculados do seguinte modo

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

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

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

### 2.1.3 *Model-based diagnosis*

O princípio base do diagnóstico baseado no modelo (*Model-based diagnosís*) é o de comparar o modelo, isto é a descrição de funcionamento do sistema, ao comportamento efetivamente observado [**?**]. Sendo depois a diferença entre os dois usada para identificar os componentes que possam explicar os erros. Isto na prática requer uma descrição formal do sistema, o que torna a tarefa bastante difícil [**?**].

De forma a facilitar o uso deste método recorre-se por vezes à inferência do modelo, através do próprio *software*, mais especificamente através dos testes definidos neste [**?**].

Apesar da elevada fiabilidade dos resultados que resultam desta técnica, o esforço computacional necessário na criação do modelo de uma programa de grande dimensão impede, maior parte das vezes, o seu uso em projetos reais [**?**].

### 2.1.4 *Barinel*

O *Barinel* é um algoritmo que se inspira nos dois métodos descritos anteriormente, *program-spectra based* e *model-based diagnosis*, e que com isto consegue melhores resultados que as outras soluções com um custo pouco superior [**?**].

O algoritmo começa por analisar uma *hit-spectra matrix*, que representa os testes executados em relação aos componentes que foram executados e ao seu resultado final.

| | *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*

Na tabela 2.2, temos identificados 3 componentes distintos ($c_1$, $c_2$ e $c_3$), 4 testes executados ($t_1$, $t_2$, $t_3$ e $t_4$) e o respectivo resultado da execução ($e$). O valor 1 em qualquer uma das colunas das observações (*obs*) indica que o dado componente foi executado nesse teste e o valor 0 indica o contrário, que o componente não foi executado. Na coluna $e$, o algarismo 1 declara que o teste correspondente falhou. Pelo que, por exemplo, o teste $t_4$ executou os componentes $c_1$ e $c_3$ e foi concluído com sucesso.

#### 2.1.4.1 Geração de candidatos

Com base nesta matriz, uma lista de conjuntos de candidatos ($d$) é gerada, sendo esta reduzida ao número mínimo de candidatos possível, dado que estes subsumem todos os outros candidatos e minimizam o espaço de pesquisa. Este problema denominado *minimal hitting set* (MHS) é só por si um problema *NP-hard* o que levou à necessidade de criação de heurísticas próprias [**?**, **?**]. O algoritmo usado pelo *Barinel* na resolução deste problema é o *Staccato*.

Neste caso, seriam gerados apenas dois candidatos:

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

#### 2.1.4.2 Ordenação de candidatos

Para cada candidato $d$, é calculada a probabilidade de acordo com a regra de *Naïve Bayes*:

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

$Pr(obs_i)$ é apenas um termo normalizador idêntico para todos os candidatos, pelo que não é usado para proceder à ordenação.

Sendo $p_j$ a probabilidade à *priori* do componente $c_j$ originar uma falha, também denominada *prior*, podemos definir $Pr(d)$, probabilidade do candidato ser responsável pelo erro, não tendo em conta evidências adicionais, como

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

Sendo $g_j$ (*component goodness*) a probabilidade do componente $c_j$ executar de forma correta dado o facto de $c_j$ integrar o conjunto de componentes faltosos, temos que

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

Tendo em conta o nosso exemplo

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

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

Quando existem valores $g_j$ desconhecidos, é maximizado o valor de $Pr(obs, e|d)$ usando o algoritmo *Maximum Likelyhood Estimation* (MLE).

Neste caso, todos os valores de $g_j$ são desconhecidos. Executando o algoritmo MLE para ambas as funções e calculando o resultado final temos que:

- $Pr(d_1, obs, e) = 1.9 \times 10^{-9}$ ($g_1 = 0.47$ 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], anteriormente conhecido como GZoltar[2], é a ferramenta que materializa o algoritmo *Barinel* e que permite uma análise de projetos *Java* [**?**].

Através de injeção de código e da execução dos testes *JUnit*, o *Crowbar* é capaz de identificar os componentes que foram executados e associá-los aos testes falhados.

A representação dos resultados pode ser feita de várias formas, sendo a principal a visualização *Sunburst* que podemos ver na Figura 2.1 e que apresenta em cada anel um grau de granularidade diferente, desde o projeto à linha de código [**?**].

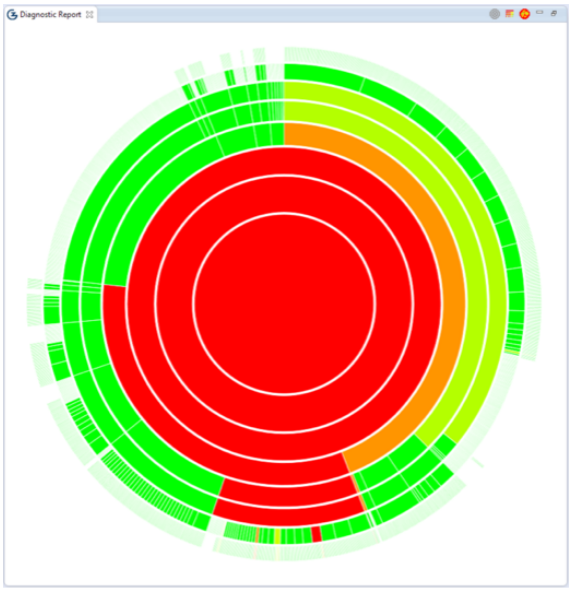


Figure 2.1: Visualização Sunburst do Crowbar

Para além desta informação, a ferramenta apresenta-nos também com a lista de componentes ordenada de acordo com a probabilidade de este ter falhas.

## 2.2 *Software Repository Mining*

Entende-se por *Software Repository Mining* todo o *software* que é capaz objetivo de extrair dados de repositórios, como *Git* ou *SVN*.

### 2.2.1 Identificação de correções e defeitos

Na extração de informações de repositórios de código, como o *Git*, torna-se importante identificar o tipo de alterações feitas. A descrição textual associada à alteração é bastante importante nesta

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

identificação e permite fazê-lo com um elevado grau de precisão, dependendo do projeto em causa [**?**].

É estimado que 34 a 46% de todas as alterações correspondam a correções e que estas representem 18 a 27% das linhas adicionadas e removidas [**?**]. Foram também encontrados padrões que relacionam as correções com o seu tamanho, em termos de código, e com o dia da semana em que foram executadas [**?**].

Após esta identificação é possível identificar as alterações que introduziram os erros usando o algoritmo *SZZ* [**?**]. Este algoritmo recorre ao histórico e à análise do código em questão para corretamente localizar o *commit* (alteração) que introduziu o defeito, ignorando, por exemplo, alterações que apenas introduzam comentários.

### 2.2.2 Ferramentas

Existem diversas ferramentas que facilitam este processo, pelo que destacamos duas.

#### 2.2.2.1 *libgit2*

Libgit2[3] é uma biblioteca multi-plataforma, sem dependências, que permite a interação direta com o repositório *Git* com uma performance nativa.

Esta biblioteca pode ser usada em qualquer linguagem que permita ligação a *C*, pelo que existem dezenas de implementações em linguagem como *Python*[4], *Javascript*[5] e *Java*[6].

#### 2.2.2.2 *GHTorrent*

GHTorrent[7] é um projeto que tem por objetivo criar um armazém de dados escalável, facilmente pesquisável, mesmo *offline*, que replique a informação obtida através da API REST do GitHub[8].

Esta ferramenta funciona de uma forma distribuída e monitoriza a API do GitHub através da página `https://api.github.com/events`. Cada evento, despoleta a extração de informação e conteúdo para uma base de dados *MongoDB* e outra *MySQL* [**?**].

## 2.3 Abordagens à previsão de defeitos

Foram já exploradas algumas abordagens para a predição da qualidade do código. Salientam-se neste capítulo algumas técnicas como *BugCache*, *FixCache* e *Buggy Change Classification*, que são técnicas que se incluem no tipo de abordagem que pretendemos seguir (*change log*) e que permitem, em conjunto com outras técnicas, otimizar a localização dos defeitos.

[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/`

### 2.3.1 *BugCache/FixCache*

Com base na premissa de que as falhas nunca ocorrem isoladamente e que portanto onde existe um defeito existem outros, o *BugCache* cria uma lista de componentes onde a probabilidade de conterem erros é elevada, com base na análise em todo o histórico de alterações do projeto.

Este algoritmo destaca-se pela sua precisão, conseguindo uma exatidão de 73 a 95% quando usado com uma granularidade ao nível do ficheiro, sendo portanto o melhor até à data [**?**].

Os defeitos são identificados por ordem temporal e adicionados à lista. Quando esta lista atinge o tamanho máximo, os componentes vão sendo removidos de acordo com o método de substituição escolhido. Existem diversos métodos que utilizam diferentes métricas como última utilização (*Least Recent Used - LRU*), número de defeitos recentes e número de alterações recentes [**?**].

Conclui-se com este algoritmo que [**?**]:

- Caso tenha sido introduzido um defeito, há uma tendência para serem introduzidos mais defeitos em breve. (*temporal locality*)
- Caso um componente tenha sido acrescentado ou modificado recentemente este tem uma maior probabilidade de ser defeituoso (*changed-entity locality*, *new-entity locality*)
- Caso um componente tenha introduzido um erro, os componentes ligados mais diretamente a este introduzirão também erros no futuro (*spatial locality*)

A diferença entre o *BugCache* e o *FixCache* é o momento em que cada um atualiza a lista. Sendo que o primeiro atualiza a lista quando um defeito é introduzido, o segundo atualiza apenas quando o erro é corrigido. Devido a esta diferença a implementação do *FixCache* torna-se mais fácil.

### 2.3.2 *Buggy Change Classification*

*Change Classification* tem uma abordagem fundamentalmente diferente e tem um objetivo também distinto. O *Change Classification*, com recurso a técnicas de *Machine Learning* e às informações disponíveis quanto a erros anteriores, é capaz de prever se uma alteração introduziu ou não um novo defeito. Esta previsão é feita com um rigor muito elevado de 78% [**?**].

Numa primeira fase, todas as alterações feita até ao momento são classificados como *buggy* ou *clean*. Com esta classificação feita e com a extração de dados para cada também concluída, procede-se à criação de um modelo através do treino de um algoritmo de classificação.

O tipo de dados usados dividem-se em sete grupos: métricas de complexidade, código adicionado, código removido, nome do ficheiro e diretório, novo código e meta-dados [**?**].

Tanto *Support Vector Machine* (SVM) como *Naïve Bayes* foram usados no estudo, tendo sido o classificador feito com recurso a SVM aquele que apresentou melhores resultados [**?**].

[TODO] Talk about the Amir Elmishali's paper

[TODO] Add Background about Machine Learning?

# Chapter 3

# Estimating Defect Probabilities

In order to estimate the defect probabilty 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 as 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 probabily 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 metadata 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, must extract metadata knowledge from past commits with faulty components, the parent commits of a fix. For each of these commits, all of components should be 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 can not used. The only information that must be used is information diretly related to the file changes, authors, number of lines or bytes.

Along with the extraction of the metadata information from the past faulty states, it must also extract the metadata 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 present in the current state of the project.

[TODO] Image

## 3.2 Install

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

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

## 3.3 Usage

After all the 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:

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

It's possible to specify a diferent 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's 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's message matches the defined regex, doesn't contain any word from the stop words list, such as "typo" or "javadoc", and it isn't a merge, adds it to the list of commits to analyze.

[TODO] Regex

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 it's 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

Since this procedure has a high computation cost, some enhancements have been made. It relies heavily on *Redis* for caching and the number of components analyzed is reduced when possible. Considering that the modeling step will balance the training data, reducing the number of *clean* components according to the number of existing *buggy* components, the system randomizes the list and limits the extraction of *clean* components up to a maximum of four times the number of *buggy* components on the given commit.

[TODO] Formula representing the maximum number of components

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

### 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

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.

Ploting 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.

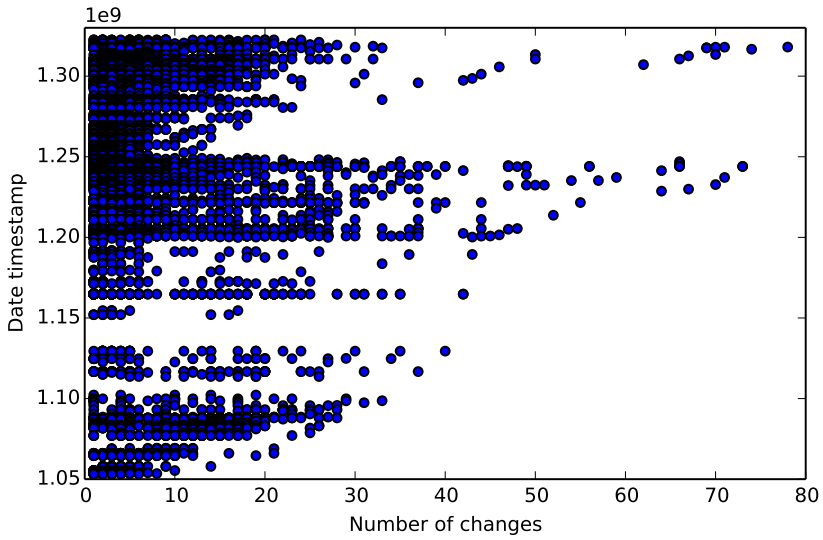


Figure 3.1: Example of the influence of the date on number of changes. (Project Math, Defect 1)

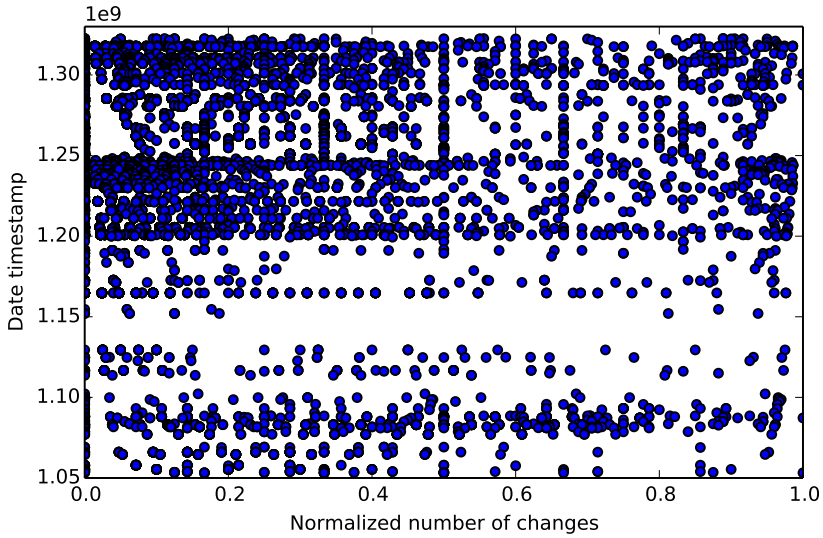


Figure 3.2: Example of the relation between the date on the normalized number of changes. (Project Math, Defect 1)

```
1 [
2   {
```

```
    "__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,
    "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,
    "authorChanges:date-weighted::luc@apache.org:normalized":
        0.00025474141229450394,
    "authorChanges:size-weighted::luc@apache.org:raw": 33,
    "authorChanges:size-weighted::luc@apache.org:normalized": 0.5454545454545454,
    "authorChanges:date+size-weighted::luc@apache.org:raw": 32.774417,
    "authorChanges:date+size-weighted::luc@apache.org:normalized":
        0.5455704368440787,
    ...
},
```

```
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's presented in the Experimental Results chapter **??**. Neural Networks weren't 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 refered 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

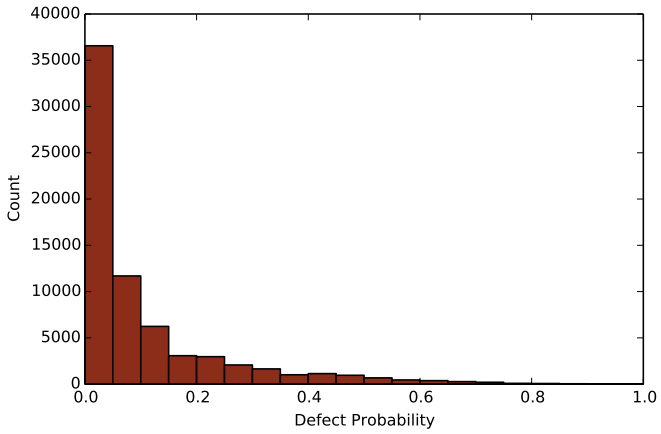


Figure 5.1: Distribution of predicted defect probabilities for probably clean components

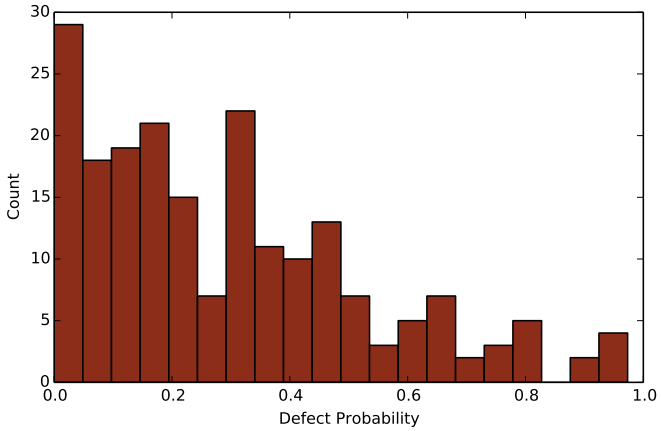


Figure 5.2: Distribution of predicted defect probabilities for faulty components

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

Figures 5.1 and 5.2 illustrate the different defect probabilities distributions of faulty and probably clean components, across all the 184 defects, respectively. 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 5.3 illustrates a case where the faulty component has a low defect probability, 12%, but it still is on the top 17%.

Figure 5.4 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%.

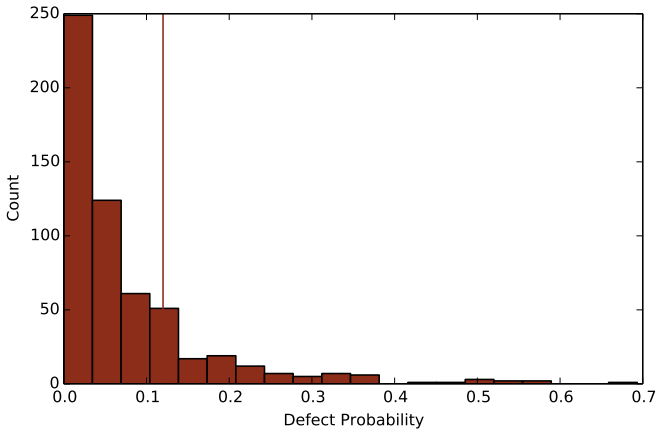


Figure 5.3: Histogram of predicted defect probabilities for Project Math (Defect 50), with a vertical line identifying the faulty component prediction

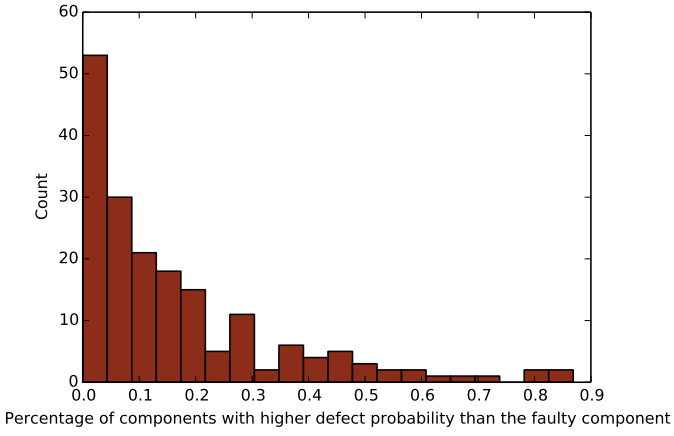


Figure 5.4: Histogram of the percentage of components with higher defect probability than the faulty component for all 184 defects

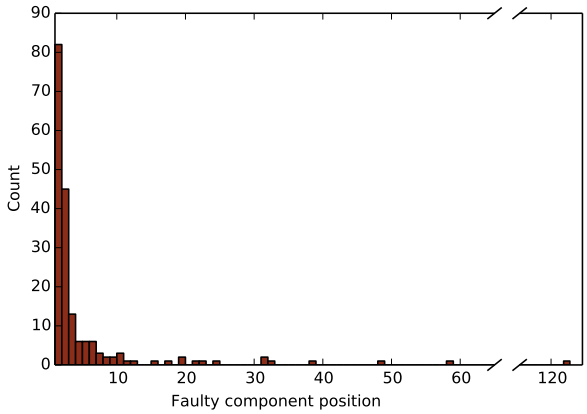


Figure 5.5: Histogram of the relative position of all faulty components, with a probability above zero, reported by the unmodified barinel

## 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 5.5, 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 won't consider on the graphics 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 improve 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 5.6.

### 5.2.2 Priors Replacement

[TODO] ... Waiting ...

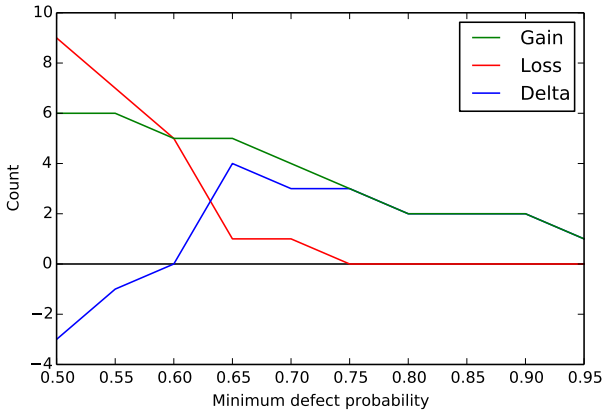


Figure 5.6: Results modification effect on Barinel results, by minimum defect probability

# Chapter 6

# Discussion

The experimental results acknowledged that extracting the component changes metadata is valuable by allowing to predict, with good precision and independently of the language, which files are more probable to have defects (Figure 5.4).

## 6.1 Estimating Defect Probability

In figures 5.1 and 5.2 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, being the reverse of figure 5.1, but this may have been affected 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 be defectuous 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.

Further more, one important aspect that can be observed on figure **??** and was verified by experimentation is that the model that allows to predict the faulty components now may not work in the future. The model must be trained with the most recent data, because the patterns that allow to identify to distinguish clean and fault component appear to change along with the project.

Yet, even with so much aspects that may affect our prediction accuracy, the figure 5.4 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 5.5, showed how good the results are and the tenuous percentage of tests that can be improved by using our approach to modify Barinel results. 14.67% in the best case scenario. While in the worst case scenario, 29% of the tests would worsen.

Figure 5.6 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 occured 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.

Given this data, we can concluded with some confidence that components with a predicted defect probability above 0.65 have in fact a high real defect probability.

[TODO] The other barinel integration

## 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 behaviour 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.

# Chapter 7

# Conclusions and Further Work

As said before, we consider that building tools to help developers do a better job is crucial and have a huge impact on the global economy. With this research we make one more step in the right direction.

The data mining application developed is based only on metadata information from the files on the Git repository, what makes it language agnostic, and showed that is capable to identify components with an high defect probability, the code hotspots

This capacity of identifying hotspots is highly valuable and can save help save time. It can be used, for example, to advise software developers doing code reviews to be more careful about some file, or to improve results from Fault Localization Software, such as Barinel .

## 7.1 Goals Satisfaction

The application is able to to predict the defect probability for any component on any software project, that uses Git, with a precision that also allowed to improve some results of Barinel.

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

## 7.2 Further Work

This project can be further improved and can also be used to create new or integrate with existing applications for which defect probability is important. There are plenty relevant opportunities to further work:

- Improving fix commits identification
- Improving precision by adding static code analysis features
- Improving machine learning model accuracy, by reducing the noise, removing outlier and better managing the unbalanced data

- Integrating with GitHub Pull Requests to help identify code hotspots
- Improving extracting performance

[TODO] Add more and explain each deeply. Maybe add a image of the possible GitHub integration?

# 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.

[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.

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

[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.

REFERENCES


[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.