UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

RENAN GOMES VIEIRA

CONTRIBUTIONS TO BUG-FIXING TIME ESTIMATION: AN EMPIRICAL STUDY IN OPEN SOURCE PROJECTS OF APACHE ECOSYSTEM

FORTALEZA
2022

RENAN GOMES VIEIRA

CONTRIBUTIONS TO BUG-FIXING TIME ESTIMATION: AN EMPIRICAL STUDY IN OPEN SOURCE PROJECTS OF APACHE ECOSYSTEM

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de doutor em Ciência da Computação. Área de Concentração: Aprendizado de Máquina e Engenharia de Software.

Orientador: Prof. Dr. João Paulo P. Gomes.

Coorientador: Prof. Dr. Lincoln Souza Rocha.

FORTALEZA

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

V718c Vieira, Renan Gomes.
Contributions to bug-fixing time estimation : an empirical study in open source projects of Apache ecosystem / Renan Gomes Vieira. – 2022.
120 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2022.
Orientação: Prof. Dr. João Paulo Pordeus Gomes.
Coorientação: Prof. Dr. Lincoln Souza Rocha.

1. Relatórios de bug. 2. Aprendizado de máquina. 3. Estimativa de tempo de correção. 4. Análise Bayesiana de dados. I. Título.

CDD 005

RENAN GOMES VIEIRA

CONTRIBUTIONS TO BUG-FIXING TIME ESTIMATION: AN EMPIRICAL STUDY IN OPEN SOURCE PROJECTS OF APACHE ECOSYSTEM

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de doutor em Ciência da Computação. Área de Concentração: Aprendizado de Máquina e Engenharia de Software.

Aprovada em: 26/05/2022

BANCA EXAMINADORA

Prof. Dr. João Paulo P. Gomes (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Lincoln Souza Rocha (Coorientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. César Lincoln Cavalcante Mattos
Universidade Federal do Ceará (UFC)

Prof. Dr. Ajalmar Rêgo da Rocha Neto
Instituto Federal do Ceará (IFCE)

Prof. Dr. Pedro de Alcântara dos Santos Neto
Universidade Federal do Piauí (UFPI)

AGRADECIMENTOS

Aos meus pais e irmãos, Dinha, Valmir, Lucas e Julyanna, pelo apoio incondicional durante todos esses anos de formação acadêmica. O suporte e presença de vocês é fundamental em tudo que busco realizar em minha vida.

Aos meus orientadores, João Paulo e Lincoln, pela parceria durante esses anos de doutorado. Ao mesmo tempo que fui guiado através de nossas reuniões, discussões e aulas, sempre me senti livre e incentivado para explorar minhas próprias ideias, o que foi determinante para meu desenvolvimento enquanto pesquisador. Grato pela oportunidade, disponibilidade, confiança e conhecimento compartilhado.

Aos colaboradores de pesquisa: professor César Lincoln, sempre prestativo e atencioso, que contribuiu de forma significativa em diversas etapas desse projeto; Diego Parente, que me auxiliou de forma generosa nas pesquisas relacionadas a inferência bayesiana; e aos pesquisadores Matheus Paixão e Ricardo Britto, que colaboraram na escrita, revisão e concepção de dois dos artigos científicos desenvolvidos durante o doutorado.

Aos colegas do MDCC: Bustamante, Daniel, Diego Farias, Ernando, e Thiago pelas diversas discussões (nem sempre) acadêmicas, litros de café compartilhados e momentos de descontração durante todos esses anos.

Aos diversos amigos e amigas, em especial, Antônio, Dennis Sávio, Eudenia, Madson, e Milaynne. Os inúmeros momentos de companheirismo através de conversas, compartilhando experiências ou amenidades, tornaram essa jornada mais leve, principalmente nos seus momentos mais turbulentos.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001

Meu muito obrigado a todos e todas!

RESUMO

A correção de *bugs* é um aspecto crucial da manutenção de *software*. Desenvolvedores e gerentes precisam lidar com relatórios de *bugs* que precisam de atenção imediata, apesar dos recursos limitados. Geralmente, projetos de *software* usam sistemas de rastreamento de *issues* como uma forma de relatar e monitorar tarefas de correção de *bugs*. Essas fontes de dados tem sido utilizadas por pesquisadores para conduzir estudos e melhor entender o problema, fornecendo meios para reduzir custos e aumentar a eficiência na tarefa de correção. Esta tese apresenta três contribuições para o processo de correção de *bugs*. A primeira é um conjunto de dados e o seu *script* de mineração, junto a uma série de análises e visualizações. Descrevemos o processo de aquisição, a necessidade de minerar um novo conjunto de dados, além de uma análise sobre alguns campos de relatórios que usamos nas subsequentes contribuições desenvolvidas. A segunda contribuição é uma nova abordagem para estimar o tempo de correção do *bugs*, onde consideramos o conceito de evolução do relatório de *bug*. Primeiro, verificamos com que frequência os relatórios de *bug* e seus campos são atualizados. A seguir, avaliamos a abordagem usando diferentes métodos de classificação de aprendizado de máquina, com distintas configurações de saída e técnicas de balanceamento de classes. Utilizando os melhores modelos testados para os diferentes estágios da evolução de um relatório, avaliamos se existem diferenças na capacidade de estimativa dos modelos segundo o estado de um relatório. Reunimos evidências de que os campos dos relatórios são atualizados com frequência, caracterizando sua a evolução, impactando nas estimativas dos modelos de predição de tempo de correção. A avaliação dos modelos mostra resultados promissores ao predizer se um *bug* será corrigido em menos ou mais de cinco dias, especialmente nos estados iniciais dos relatórios. A terceira contribuição é um estudo sobre a relação entre o tempo de correção de *bug* e três campos: prioridade, *links* (relação entre relatórios) e *code-churn* (relacionado ao *patch* de correção do bug). Através de análise Bayesiana de dados, avaliamos dois modelos diferentes - um ‘específico’ para cada conjunto de dados e um ‘hierárquico’ considerando todos os projetos de uma vez. Outros três modelos hierárquicos são explorados como forma de ilustrar a flexibilidade deste tipo de modelagem. Reunimos evidências de que relatórios de *bug* com *links* e valores maiores de *code-churn* demandam mais tempo para serem corrigidos, ao contrário de prioridade que não apresenta influência no tempo de correção.

Palavras-chave: relatórios de *bug*; aprendizado de máquina; estimativa de tempo de resolução; *Jira Issue Tracking System*; análise Bayesiana de dados.

ABSTRACT

Fixing bugs is a crucial aspect of software maintenance. Developers and managers must deal with many bug reports that need immediate attention despite limited resources and tight deadlines. Generally, software projects use issue tracking systems to report and monitor bug-fixing tasks. Several researchers have used this data source to conduct research and better understand the problem, providing means to reduce costs and improve efficiency in the correction task. This thesis presents three contributions to the bugs correction process. The first is a dataset and its mining script, along with a series of analyzes and visualizations. We describe the data acquisition process, the necessity to mine a new dataset, and provide a deeper analysis of some reporting fields that we use in the subsequent contributions presented in this thesis. A second contribution is a new approach to estimating the time to fix bugs. We consider the concept of bug report evolution to create a dataset containing all investigated report states. First, we check how often the bug reports and their fields are updated. Next, we evaluate our approach using different machine learning methods as a classification problem, with a number of output configurations and class balancing techniques. Using the best models (considering all possible designs) for the different stages of the evolution of a bug report, we evaluate whether there are significant differences in the estimation capacity of the models according to the report state. We gathered evidence that report fields are frequently updated, which characterizes the evolution of reports, impacting the creation of bugs fixing-time estimation models. The evaluation of the models shows promising results in predicting whether a bug will be fixed in less or more than five days, especially in the initial states of the reports. The third contribution is a study on the relationship between bug correction time and three fields: priority, links (the relationship between reports), and code-churn (related to the fixing patch associated with the bug report). Through Bayesian data analysis, we evaluated two different models - one ‘specific’ for each project and one ‘hierarchical’ considering all projects at once. We also explored three other hierarchical models to illustrate the flexibility of this type of modeling. Finally, we have gathered evidence that bug reports with links and higher values of code-churn (above the project’s median) tend to take longer to fix. On the other hand, the priority level appears to have no significant influence on the time to fix a bug.

Keywords: bug report; machine learning; resolution time estimation; Jira Tracking Issue System; Bayesian data analysis.

LIST OF FIGURES

Figure 1 – Customized Jira issue workflow. 20
Figure 2 – Log-transformed bug-fixing time boxplot by project. 26
Figure 3 – Log-transformed bug-fixing time boxplot by category. 26
Figure 4 – Priority distribution. 27
Figure 5 – Example of links in bug reports. 28
Figure 6 – The proportion of different scenarios of links 29
Figure 7 – The number of reports updates histogram 32
Figure 8 – The number of status changes by reports. 34
Figure 9 – The reports states re-creation process. 40
Figure 10 – Bug reports resolution time calculation. 41
Figure 11 – The train/test 5-fold split method. 52
Figure 12 – Workflow to report RRT evaluation by progress and interval. 59
Figure 13 – Accuracy evaluation by report resolution progress. 60
Figure 14 – Accuracy evaluation by report resolution interval. 60
Figure 15 – Graph 'specific-model' representation. 77
Figure 16 – Graph hierarchical model - 'HM-AP' representation. 78
Figure 17 – μ posterior distributions - 'specific-models', 'links' results. 80
Figure 18 – μ_0 posterior distributions - 'HM-AP', links results. 81
Figure 19 – μ posterior distributions - 'specific-models', 'priority' results. 82
Figure 20 – μ_0 posterior distributions - 'HM-AP' model, 'priority' results. 84
Figure 21 – μ posterior distributions - 'specific-models', 'code-churn' results. 85
Figure 22 – μ_0 posterior distributions - 'HM-AP', 'code-churn' results. 86
Figure 23 – Graph hierarchical model - 'HM-$\mathcal{G}$' representation. 89
Figure 24 – μ_0 posterior distributions - 'HM-$\mathcal{G}$', 'links' results. 90
Figure 25 – μ posterior distributions - 'specific-models', 'links' results (55 projects). . . . 106
Figure 26 – μ posterior distributions - 'specific-models', 'priority' results (55 projects). . 109
Figure 27 – μ posterior distributions - 'specific-models', 'code-churn' results (55 projects).113
Figure 28 – Prior predictive. 116
Figure 29 – Posterior predictive. 117
Figure 30 – μ_0 posterior distributions - 'HM-$\mathcal{G}$', 'priority' results 118
Figure 31 – μ_0 posterior distributions - 'HM-$\mathcal{G}$', 'code-churn' results. 119

LIST OF TABLES

Table 1 – List of mined projects data. 23
Table 2 – Interval bug-fixing time distribution by category. 25
Table 3 – Most common fields updates in bug reports. 30
Table 4 – Bug reports ‘status’ values. 32
Table 5 – Adjacent matrix of status changes. 33
Table 6 – Attributes used by Zhang *et al.* (2013). 43
Table 7 – Description of the two ‘sets’ of attributes used in the baseline approach. 44
Table 8 – Filtered dataset information. 47
Table 9 – Dataset features description. 48
Table 10 – Labels distribution. 50
Table 11 – Most common bug reports’ fields updates (10 projects). 52
Table 12 – Baseline results in different data scenarios: attributes Set 1. 54
Table 13 – Baseline results in different data scenarios: attributes Set 2. 55
Table 14 – All projects overall best results. 57
Table 15 – Models results classifying initial states reports. 62
Table 16 – μ posterior distribution summary, ‘specific-models’, ‘links’ results. 80
Table 17 – μ_0 posterior distribution summary. ’HM-AP’, ‘links’ results. 81
Table 18 – μ posterior distribution summary, ‘specific-models’, ‘priority’ results. 83
Table 19 – μ_0 posterior distribution summary, ‘HM-AP’ model, ‘priority’ results. 83
Table 20 – μ posterior distribution summary, ‘specific-models’, ‘code-churn’ results. . . . 85
Table 21 – μ_0 posterior distribution summary, ‘HM-AP’ model, ‘code-churn’ results. . . . 86
Table 22 – μ_0 posterior distributions summaries, ‘HM-$\mathcal{G}$’ models, ‘links’ results. 90
Table 23 – Static dataset fields - Jira. 102
Table 24 – Static dataset fields - Git 103
Table 25 – Changelog dataset fields 103
Table 26 – Comment-log dataset fields 104
Table 27 – Commit-log dataset fields 104
Table 28 – μ posterior distribution summary, ‘specific-models’, ‘No Links’ results (55 projects). 107
Table 29 – μ posterior distribution summary, ‘specific-models’, ‘With Links’ results (55 projects). 108

Table 30 – μ posterior distribution summary, 'specific-models', 'Low Priority' results (55 projects). 110
Table 31 – μ posterior distribution summary, 'specific-models', 'Medium Priority' results (55 projects). 111
Table 32 – μ posterior distribution summary, 'specific-models', 'High Priority' results (55 projects). 112
Table 33 – μ posterior distribution summary, 'specific-models', 'Lower Code-Churn' results (55 projects). 114
Table 34 – μ posterior distribution summary, 'specific-models', 'Higher Code-Churn' results (55 projects). 115
Table 35 – μ_0 posterior distribution summary from alternative hierarchical models (HM-$\mathcal{G}$), 'priority' results. 119
Table 36 – μ_0 posterior distribution summary from alternative hierarchical models (HM-$\mathcal{G}$), 'code-churn' results. 120

LIST OF ABBREVIATIONS AND ACRONYMS

ACC	Accuracy
ASF	Apache Software Foundation
AUC	Area Under the ROC Curve
BDA	Bayesian Data Analysis
BFT	Bug-Fixing Time
BoW	Bag of Words
CC	Cluster Centroids
CD	Creation Date
CI	Confidence Interval
CV	Cross-Validation
DRT	Defect Resolution Time
F1	F1-Score
HM	Hierarchical Model
HMM	Hidden Markov Model
ITS	Issue Tracking System
KNN	K-Nearest Neighbors
LGL	Log-Loss
LUD	Last Update Date
MAE	Mean Absolute error
MCMC	Monte Carlo Markov Chain
MLP	MultiLayer Perceptron
MRE	Mean Relative Error
NB	Naive Bayes
OD	Original Data
PRC	Precision
RCL	Recall
RD	Resolution Date
RND	Random Under-Sampling
RRT	Report Resolution Time
SMOTE	Synthetic Minority Over-sampling Technique

CONTENTS

1 **INTRODUCTION** 12
1.1 **Motivation** 13
1.1.1 *Dataset Mining* 13
1.1.2 *Bug-fixing time estimation* 14
1.1.3 *Bayesian data analysis on relation between bug report features and bug-fixing time* 16
1.2 **Objectives** 17
1.3 **Publications** 18
1.4 **Outline** 18
2 **A NEW APACHE BUG-FIXING DATASET** 19
2.1 **Dataset Preliminaries** 19
2.1.1 *Apache Software Foundation and Jira* 19
2.2 **Data Collection Methodology** 21
2.3 **Dataset Description** 22
2.3.1 *Static Perspective* 22
2.3.2 *Dynamic Perspective* 22
2.4 **Dataset Characterization** 24
2.4.1 *Bug-Fixing Time* 24
2.4.2 *Priority* 25
2.4.3 *Links* 27
2.4.4 *The Changelog Dataset* 30
2.4.5 *Reports Updates* 31
2.4.6 *Status Changes* 31
2.5 **Dataset Relevance** 34
2.6 **Related Work** 35
3 **THE ROLE OF BUG REPORT EVOLUTION IN RELIABLE FIXING ESTIMATION** 37
3.1 **Materials and Methods** 37
3.1.1 *A Temporal Dataset of Bug-Fixing Activities and Reports* 38
3.1.2 *Bug Reports' Fields Updates and Zhang et al. (2013)'s Work Replication* . . 42
3.1.3 *Preprocessing steps on the 'Temporal Dataset' to apply our approach* 45

3.1.4	*Models training methodology*	47
3.1.5	*The Train/Test Split Method*	50
3.2	**Results**	51
3.2.1	*Field Changes Analysis and (ZHANG et al., 2013) replication (baseline)*	51
3.2.2	*Training models with all bug reports states*	56
3.2.3	*Models Performance by Group: Progress and Resolution Intervals*	58
3.3	**Discussion**	63
3.4	**Threats to Validity**	65
3.5	**Related Works and Comparison**	66
4	**BAYESIAN DATA ANALYSIS APPLIED TO BUG REPORTS DATA**	71
4.1	**Bayesian Data Analysis**	71
4.1.1	*Bayes in a Nutshell*	72
4.1.2	*Hierarchical Models*	74
4.2	**Selected Features**	75
4.3	**Modeling Process and Models Description**	76
4.4	**Results**	78
4.4.1	*Links*	79
4.4.2	*Priority*	82
4.4.3	*Code Churn*	84
4.5	**Exploring different Hierarchical Models**	87
4.6	**Discussion**	90
4.7	**Threats to the Validity**	91
4.8	**Related Works**	92
5	**CONCLUSION AND FUTURE WORKS**	94
	BIBLIOGRAPHY	96
	APPENDIX A–DATASET FEATURES TABLE	102
	APPENDIX B–COMPLETE ‘SPECIFIC-MODEL’ RESULTS	105
	APPENDIX C–PREDICTIVE CHECK	116
	APPENDIX D–ALTERNATIVE HIERARCHICAL MODELS	118

1 INTRODUCTION

End-users and companies widely adopt open-source software (HAUGE *et al.*, 2010; LENARDUZZI *et al.*, 2020). As they grow in size and complexity to meet new requirements and needs, the goal of software quality assurance becomes increasingly more challenging. In an open-source or corporative context, software developers and engineers use several tools to improve software development. One of the most commonly used tool (SERRANO; CIORDIA, 2005; BAYSAL *et al.*, 2013) is the Issue Tracking System (ITS), a platform where any software issue [1] can be registered and traced. There are many ITSs available, namely Bugzilla, YouTrack, and Jira, among others.

Bug — a colloquial therm for errors or vulnerabilities in software systems (GOUES *et al.*, 2021) — is a particular type of issue that can hinder software quality. They can be resource-consuming, leading to costs by order of billions per year and taking on average 50% of the software developers' time for finding and fixing them (BRADY, 2013; GOUES *et al.*, 2021). The cost related to bugs are high, not just because finding and fixing faults increases the development and testing cost, but also because of the consequences of field failures due to these bugs (HAMILL; GOSEVA-POPSTOJANOVA, 2017). The bug-fixing activity is one of the most resource-intensive tasks, and this problem is worsened in large software systems. In these cases, the number of bug reports can exceed the available project resources (KARIM *et al.*, 2017). Besides the bug being a problem by itself, the whole process of triaging the bugs to be fixed is also a time-consuming task. Many questions have been raised regarding bug issues on ITS for a newly registered bug report, such as "was this bug already registered?" (LAZAR *et al.*, 2014; EBRAHIMI *et al.*, 2019), "who is the best person to fix this bug?" (GUO *et al.*, 2011; SHOKRIPOUR *et al.*, 2015), "is this a real bug?" (HERZIG *et al.*, 2013), "is this report good and does it have enough information?" (ZIMMERMANN *et al.*, 2010), "what is its priority?" (TIAN *et al.*, 2015), and "how much time is necessary to fix this bug?" (ZHANG *et al.*, 2013; AL-ZUBAIDI *et al.*, 2017; HABAYEB *et al.*, 2018).

For software that uses an ITS, bug identification is generally recorded at the ITS itself. Next, a bug triage happens, mainly being a manual[2] collaborative step. In the triage, a bug report will be examined to (i) indicate whether the report contains sufficient or duplicated information, (ii) assign the bug's severity and priority, and (iii) define who will be the person

[1] An issue can represent a story, a bug, a task, or another issue type in the project.
[2] In Mozilla's (Firefox) case, it is partly automated, see https://hacks.mozilla.org/2019/04/teaching-machines-to-triage-firefox-bugs/

responsible for fixing the bug (ARDIMENTO *et al.*, 2016), also known as the *assignee*. Other steps can be applied, such as identifying the component or version of the software affected by the bug, which can occur between or after the presented ones. Once the assignee proposes a set of patches that fixes the bug and one is selected, the patch is validated by other developers and then merged to the code, generally through a commit. The additions of screenshots, comments, and changes of reports fields are expected from the report creation until its conclusion. The record of these workflows contains information that can help to understand and improve the bug fixing process.

This thesis describes the contributions regarding the bug fixing process based on bug reports records from Jira open-source software. Most of the contributions are related to the bug fixing time estimation, but we also present contributions to the Jira data acquisition and bug fixing process.

1.1 Motivation

In this section, we provide the motivations for the three major contributions of this thesis. We present the motivation of each contribution in the chronological order of obtained results and as presented in this manuscript in the following sections.

1.1.1 Dataset Mining

Researchers use mining techniques to gather information from ITS to better understand bug reporting, triage, and fixing processes. Lamkanfi *et al.* (2013) describes the procedure to collect bug reports from Eclipse and Mozilla tracking systems. Habayeb *et al.* (2015) work proposes a Firefox bug report dataset with temporal information (i.e., report fields changes over time). The Zhu *et al.* (2016) work describes the dataset mined over Mozilla issue tracking history, and in Xu e Zhou (2018), the multi-level dataset mining process of the Linux Kernel Patchwork is shown.

Given the relevance of providing means to improve the bug fixing process, we look for structured datasets that cover most of the bug reports workflows. Generally, we found that the proposed datasets lack: i) diversity: most of the proposed datasets regarding bug reports only contain information of a few projects; ii) completeness: the whole bug fixing process is not presented. Bug reports are dynamic objects, and generally, the process description is provided in

the ITS - the general bug information that changes over time - and the version control system - the patch that fixes the bug information. None of the founded datasets provide both information.

Given this scenario, we mined a new dataset of bug-fix reports from 10 years of bug-fixing activity of 55 projects from the Apache Software Foundation (ASF). We have mined this information from the Jira issue track system concerning two different perspectives of reports with closed/resolved status: static (the latest version of reports) and dynamic (the changes that have occurred in reports over time). We also extract information from the commits (if they exist) that fix such bugs from their respective version-control system (Git). We also provide a change analysis that occurs in the reports to illustrate and characterize the proposed dataset. Once the data extraction process is a nontrivial error-prone task, we believe such initiatives could be helpful to support researchers in further, more detailed investigations. This dataset is the source for all proposals and analyses presented in this thesis.

1.1.2 Bug-fixing time estimation

Several researchers highlight the importance of providing a bug resolution time estimation. As pointed out by Al-Zubaidi *et al.* (2017), the reporters are probably interested in knowing when a particular bug will be fixed; thus, project managers may need to provide an estimation time. Such estimations can be critical to their cost planning and release management in those cases. Similarly, Habayeb *et al.* (2018) discuss that identifying bugs that would require a long fixing time right at the beginning of the bug life cycle is useful in several areas of the software quality process. This information would allow software maintenance to prioritize their work, improving the development activities on such bugs. In addition, the prediction of a bug resolution time plays a significant role in project management since it supports resource allocation and future release planning.

Some proposals use machine learning technics to solve software engineering problems. In the bug report context, there are works to estimate priority (UMER *et al.*, 2020), assignee (ALKHAZI *et al.*, 2020), bug fixing time (SHARMA *et al.*, 2019), and bug localization (RAHMAN *et al.*, 2015) of a bug report. However, different from more traditional data points observations in a machine learning dataset, bug reports are dynamics examples, i.e., attributes change over time. For instance, the reported information and the decisions made in the triage step are error-prone, demanding some update in its fields. The natural evolution of the bug fixing process requires some updates, as the study of Hu *et al.* (2014) shows that 37%-44% of

bugs have been re-assigned on bug reports of Eclipse and Mozilla, respectively. Our previous published study (VIEIRA *et al.*, 2019) shows that several changes and additions to bug reports occur during their life cycle. Our analysis show the existence of changes in a report's assignee on 54.63% of the bug reports and description modifications on 18.16%, to mention a few. We explore this through the text, but we mostly notice that these reports' fields changes and updates are overlooked in several works that use bug reports as input. Generally, the last state of a report is considered, or the research does not clearly define at what moment the reports' values are acquired.

Thus, it is evident that reports should not be seen or analyzed only when they are closed/resolved, when changes and updates in the reports may provide relevant information regarding the bug fixing process. For instance, a bug report with a high priority, with an experienced assignee associated with it and several comments and attachments, will probably take less time to be closed than one with no assignee and no comments. We argue that those reports' updates may serve as predictors regarding the reports' resolution time. Moreover, the same report in different states over its life cycle may not provide the same information about the reported bug.

When a manager opens an ITS at a particular timeline for a specific software project, the ITS may contain bug reports in several states: some recently opened, others are close to resolution. The reports also present different complexity, priority, and overall information, as the report fields are updated and changed. Thus, a tool capable of estimating the resolution time for bug reports regardless of their state in the life cycle could be highly valuable. In this thesis, one of the contributions is the investigation of the viability of providing such a tool to help software managers. In contrast to other approaches in the literature, we consider that bug reports are changeable and evolve, and such changes may impact estimation models.

Given the scenario composed of the relevance of bug report resolution estimation and the changeable and evolutionary nature of bug reports, we propose a new approach to incorporate bug reports' dynamic aspects into the predictive model's creation. We first show that the fields updates in bug reports occur and are significant; we evaluate the impact of ignoring the bug reports updates in the models and propose an approach to incorporate the reports updates into the model's training process.

1.1.3 Bayesian data analysis on relation between bug report features and bug-fixing time

The data provided in a bug report is, sometimes, the only information available to the assignee to replicate and patch the bug. Several attributes in a bug report characterize them and are used to create machine learning models, verify report quality, and better understand the bug fixing process (KARIM *et al.*, 2017; HOOIMEIJER; WEIMER, 2007). Examples of features found in the Jira ITS are: *importance-related fields* as **Priority** and **Severity**; the *text-related* as **Summary**, **Title** and **Description**; *person-related fields* as **Assignee** and **Reporter**; and *link-related fields* that deals with the relationship between reports in an ITS. The information presented in a bug report is used during the triage process to define how the development team will deal with it. Understanding how these attributes impact the quality of a bug report to help the developers in the bug fixing process is useful to build reliable estimation models and suggest bug reporting process improvements (HOOIMEIJER; WEIMER, 2007; CATOLINO *et al.*, 2019).

The workflow to understand how specific bug report characteristics are related to others that significantly impact the bug triage process (e.g., bug-fixing time, priority, or report quality) generally would use some statistical framework. Frequentist statistical approaches have been the standard tool to provide this type of insight in empirical software engineering (TORKAR *et al.*, 2021). However, the Bayesian framework is another option that sometimes is over-viewed primarily because it does not offer out-of-the-box solutions as the statistical tests of the frequentist framework. Several works have advocated using the Bayesian framework as an alternative to the more traditional statistical test use. They highlight a fine-grained data model's building control, better visual appeal of the results, and the use of additional information as prior, when compared to the frequentist framework (MCELREATH, 2020; FURIA *et al.*, 2021; GELMAN *et al.*, 2020).

The Bayesian Data Analysis (BDA) characteristics come with a cost: while frequentist statistical approaches provide a group of tests covering several data and assumptions scenarios, the Bayesian framework requires more detailed attention as we have to build our models from the bottom up. There are a few steps and attention to details to cover, and there are some works that describe the process at length in the literature, as Gelman *et al.* (2020) and McElreath (2020). In contrast, others have been more active in highlighting the advantages of using the BDA in empirical software engineering (TORKAR *et al.*, 2021; FURIA *et al.*, 2021).

Besides all the highlights provided by some researchers, one more specific objective point was crucial to adopt the Bayesian approach in this thesis: we have multiple data sources in our proposed dataset. As our intention was to provide conclusions about all the 55 projects

we mined, hierarchical models (a.k.a. multilevel models) are a potent tool provided by BDA that helps test hypotheses about the data more generalistic. The counterpoint in the frequentist statistical approaches is to combine p-values from different statistical test results from different data sources (HEARD; RUBIN-DELANCHY, 2018). As covered by Furia *et al.* (2021), there is not a uniform view about when and how the adjustment of p-values from different statistical tests. In the same work, the authors show the impact of using different techniques to adjust p-values and how they change the final conclusions regarding the analysis.

It is important to notice that both frequentist and bayesian analysis provides similar conclusions when correctly applied (FURIA *et al.*, 2021), but BDA provides a few particularities that we considerer more appealing. We summarize the motivation to apply BDA as follows:

1. Flexibility to create the models: we have total control of the assumptions regarding the model and data. As we describe every aspect of the model, this provides a better overview of the modeling process, allowing a more detailed review and criticism from the peers.
2. Hierarchical models: we mined data from several projects, each one with its particularities. Hierarchical models provide ways to summarize data from different sources to give us a more general picture of a similar behavior underlying their idiosyncratic. The use of hierarchical models serves us as an alternative to possible pitfalls of selection p-values adjustments.
3. Posterior distributions as results: the outcome of every BDA are posterior and predictive distributions. These posterior distributions describe our models' parameters based on our assumptions and data. The use of prior is an inherent characteristic of BDA. For future works that perform similar analysis on other bug reports data, our posterior distributions can be used as priors in their models, creating a chain of knowledge of the same domain (MCELREATH, 2020; FURIA *et al.*, 2021).

1.2 Objectives

The overall goal of this thesis is to provide contributions over the gaps found in the literature regarding the bug fix time estimations based on bug reports. To address these points, we define the following specific objectives that need to be achieved:

- Propose a curated dataset and the mining script to mine bug fixing activities from open-source in Jira;
- Describe the dataset with a set of visualizations and analysis;

- Show the dynamic aspect of bug reports and how they impact the reliability of bug-fixing time estimators;
- Propose a new approach to deal with the dynamic aspect of bug reports in the bug-fixing time estimators;
- Compare our proposal with previous proposals of bug-fixing time estimators.
- Describe a practical use of Bayesian statistics in bug report data to show how a set of features relates to bug fixing time.

1.3 Publications

Vieira, R. G., da Silva A., Rocha L. S., Gomes J. P. P., From reports to bug-fix commits: A 10 years dataset of bug-fixing activity from 55 apache's open-source projects. In: Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, ACM, New York, NY, USA, PROMISE'19, pp 80–89, DOI 10.1145/3345629.3345639, URL: http://doi.acm.org/10.1145/3345629.3345639

Vieira, R. G., Mattos, C. L. C., Rocha, L. S., Gomes, J. P. P., Paixão, M. H. E., (2021) The Role of Bug Report Evolution in Reliable Fixing Estimation. In: Empirical Software Engineering (Special Edition: Machine Learning Techniques for Software Quality Evaluation) (final round of revision).

Vieira, R. G., Mesquita D. P. P., Mattos, C. L. C., Britto, R. S., Rocha, L. S., Gomes, J. P. P., (2022) Bayesian Analysis of Bug-Fixing Time using Report Data. In: Empirical Software Engineering International Week, September 19–23, 2022, Helsinki, Finland, ESEM '22.

1.4 Outline

We present the thesis remainder. Chapter 2 contains the details of the acquisition process, description, and analysis of the proposed dataset. Chapter 3 presents a new approach to estimate bug fix time based on their reports, considering they natural evolutive process. Chapter 4 describes a practical use of bayesian statistics on bug reports data to relate some features and the bug fixing time. Finally, Chapter 5 concludes this thesis with final thoughts regarding the research.

2 A NEW APACHE BUG-FIXING DATASET

In this chapter, we describe the process of acquiring the dataset used in this thesis. We also provide the description and some analysis of the data.

2.1 Dataset Preliminaries

This section provides a brief introduction to Apache Software Foundation, the Jira issue tracking system, and the Jira's issue/bug-fix report life cycle. This summary is important to define more precisely our dataset, the context, and under what organizational rules these bug-fix reports are created until be considered completed.

2.1.1 Apache Software Foundation and Jira

The Apache Software Foundation (ASF) is a decentralized open-source community of developers. It is an American non-profit corporation created to support the Apache Software Projects. Every software produced by the ASF is distributed under the terms of the Apache License, being all free and open-source. On their official site[1], they list more than 350 active projects distributed over 40 different software categories. We choose projects from ASF as a target of our study because they are mature, well documented, and widely used by developers to build new systems, indicating a certain degree of reliability from the developers' perspective.

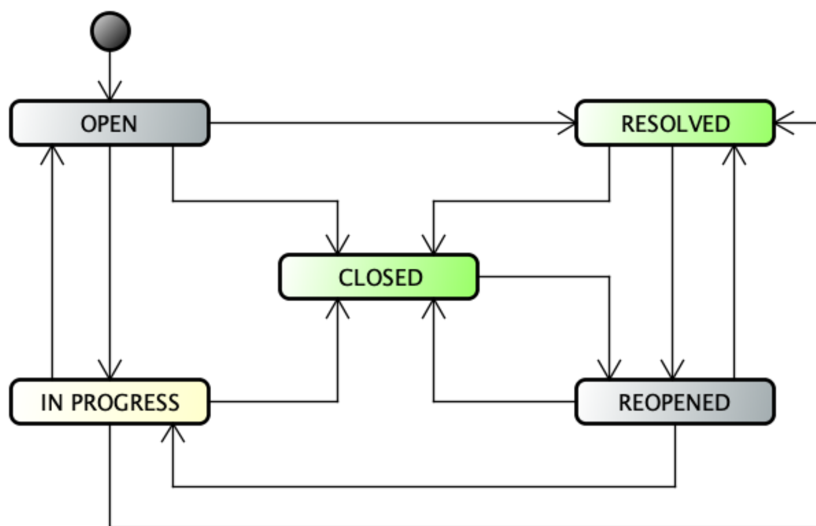
Jira[2] is a proprietary issue tracking product developed by the Australian Atlassian Corporation. For a number of open source projects, Atlassian provides the Jira services for free. By default, the ASF projects adopt Jira as their issue tracking tool. According to the Jira documentation[3], "*Issues are the building blocks of any Jira project. An issue could represent a story, a bug, a task, or another issue type in your project*". An issue has several fields[4], such as summary (a brief, one-line synopsis of the issue), description (a detailed issue explanation), priority (the importance of the issue concerning other issues), type (the type of the issue, e.g., "Bug" or "Task"), reporter (the person who created the issue), assignee (the person to whom the issue is currently assigned), watchers (number of people interested in the issue), status (the issue current state on Jira workflow), resolution (a record of the issue's resolution), created (the time

[1] https://projects.apache.org/projects.html
[2] https://atlassian.com/software/jira
[3] https://confluence.atlassian.com/jirasoftwarecloud/working-with-issues-764478424.html
[4] https://confluence.atlassian.com/adminjiracloud/issue-fields-and-statuses-776636356.html

and date on which the issue was created), and resolved (the time and date on which the issue was resolved).

Jira workflow defines a set of states and transitions that an issue passes over during its life cycle. It can be customized to meet organization needs and ensure compliance with their internal processes. Fig. 1 shows an example of customized Jira workflow, composed of 5 states (all possible statuses an issue can assume) and 12 transitions (all possible paths an issue can pass through).

Figure 1 – Customized Jira issue workflow. The Figure shows five reports states and the possible transitions between them.



To report a bug in the Jira tool, you need to set the issue field `Type` as a "Bug" during the issue creation, reaching the `OPEN` status (Fig. 1). After that, the issue can move to other states: `CLOSED`, `RESOLVED`, or `IN PROGRESS`. Reaching the `CLOSED` status, the issue is considered finished, and the value of `Resolution` field must be changed to "Fixed". Similarly, when the issue transits to `RESOLVED` status means that a resolution has been taken and is waiting for the bug reporter verification. At this point, the value of `Resolution` field can also be changed to "Fixed". The status `IN PROGRESS` means that someone is actively working to fix the bug. For some reason, an issue can move from `CLOSED` or `RESOLVED` to `REOPENED` status to perform some update or to revise the solution given to fix the bug.

2.2 Data Collection Methodology

The dataset was built through an automated mining process. We use Python 3.7.2 programming language to mine, process, and analyze the dataset. All data came from the official Jira[5] and Git[6] repositories from the ASF. First, we started mining information from Jira using Jira-Python[7], a library that facilitates the manipulation of the Jira REST API with Python. In this stage, we mine issues with type “Bug”, with `CLOSED` or `RESOLVED` status, with ”Fixed” resolution field, and that was created and fixed between 2009-01-01 and 2019-01-02.

To make feasible the tracking between a bug report and the commits performed to fix this bug, the ASF projects developers adopt, by default, the strategy of specifying the Jira issue ID in the messages of the commits devoted to resolving the issue. Thus, we use the issues' ID to track down and mine the source code changes information from commits that fix the bugs using Pydriller (SPADINI *et al.*, 2018)[8]. Then, the script creates the first dataset file we call `snapshot`. It is important to notice we verify that there are cases where there is only one commit, more than one commit, and even no commits containing a Jira issue ID in the commit message. Curiously, we discovered several cases where one commit is related to several reports. Hence, all these cases are registered on the dataset.

Next, we used the obtained issues IDs to mine all the changes made in each one of them during the considered period, creating the `change-log` dataset. After that, for each issue ID, we mined all comments posted in the report during the same period, creating the `comment-log` dataset file. Additionally, we use the issues IDs to mine all related commits, creating a dataset with detailed information regarding the bug-fixing commits we call `commit-log` dataset file. Finally, it is important to mention that we performed a preprocessing in some issue's text fields (issue summary, description, comments, and commit messages) using Python NLTK[9] library to extract and store in the dataset the top 1,000 most frequent words and its respective frequencies. All the scripts used in the mining process and the full dataset itself can be found in the replication package[10].

---

[5] https://issues.apache.org/jira
[6] http://gitbox.apache.org
[7] https://jira.readthedocs.io/
[8] https://github.com/ishepard/pydriller
[9] https://www.nltk.org/
[10] https://doi.org/10.6084/m9.figshare.8852084

2.3 Dataset Description

The dataset comprises bug reports from 55 projects of ASF. Each project has its own `snapshot`, `comment-log`, `commit-log`, and `change-log` file. These projects are distributed over nine categories: big-data (10), database (8), cloud (6), network-server (6), web-framework (6), security (3), build-management (4), library (9), and machine learning (3). The project name list is in Table 1, with additional information as the project category, year of the first release, and the number of the mined issues for each project. We collected issues regarding two perspectives: *static* and *dynamic*.

2.3.1 Static Perspective

In the static perspective, we collect information from the last version of the bug report available in the Jira tool with status `CLOSED` or `RESOLVED`. We extracted 53 attributes divided into two major groups for each bug report: the features collected from the report itself (Jira attributes) and the ones collected from the Git repository (Git attributes). For organization purposes, we also classified the dataset fields regarding the nature of the information they represent: general (standard information), text (textual information), time (time-related information), versioning (system version related information), summation (fields that store counting information), link (bug dependencies), and source (source code related information). The complete list of static perspective dataset (`snapshot`) fields can be found in Appendix A. Most of the names are self-explanatory, but a complete field description can also be found in the replication package of this chapter[11].

2.3.2 Dynamic Perspective

Every bug report starts with a few standard pieces of information provided by the author. For instance, some attributes are always created with the report and never change, such as `Key` (the report unique identifier) and `CreationDate`. Other attributes, like `Assignee` and `Priority`, are commonly defined in the report creation, but they are not mandatory and can be changed during the report lifetime. A few other attributes are hard to input unless a time has passed after the report creation. Examples are the versions affected by the bug and other related

[11] https://figshare.com/articles/dataset/Replication_Package_-_PROMISE_19/8852084/5

Table 1 – The complete list of mined projects. The table shows the category, the year of the first release, and the number of mined bugs.

Category	Project	1st Release	#Bugs
big-data (10)	Hadoop Core	2006	2861
	Hadoop Yarn	2012	2090
	Hadoop HDFS	2009	3214
	Hadoop MapReduce	2009	2210
	Flink	2015	3317
	Spark	2014	6380
	Oozie	2012	1420
	Kafka	2013	2404
	Storm	2017	1033
	Giraph	2018	373
database (8)	Hive	2010	7105
	Cassandra	2009	5001
	Lucene Core	2006	2004
	HBase	2006	6693
	ZooKeeper	2006	882
	Derby	2006	1083
	OpenJPA	2006	653
	Phoenix	2006	1564
cloud (6)	Libcloud	2011	229
	jclouds	2013	435
	Mesos	2013	2558
	VCL	2013	425
	Helix	2014	188
	Ignite	2015	2726
network-server (6)	TomEE	2013	713
	Mina	2006	171
	Mina FtpServer	2008	66
	Mina SSHD	2009	285
	Mina Vysper	2010	66
	Camel	2009	3232
web-framework (6)	MyFaces	2005	913
	Struts	2002	725
	Nutch	2012	547
	Isis	2013	410
	Solr	2006	2249
	Tapestry	2009	711
security (3)	Kerby	2006	179
	Fortress	2015	76
	Syncope	2012	654
build-management (4)	Archiva	2007	320
	Ivy	2006	204
	MNG	2012	584
	Buildr	2009	227
library (9)	Commons Compress	2009	189
	Commons Colections	2001	103
	Commons IO	2004	103
	Commons Math	2004	418
	Commons Codec	2003	64
	Commons LANG	2002	264
	Log4J 2	2014	754
	Crunch	2013	292
	Tika	2011	762
machine-learning (3)	SystemML	2017	489
	Mahout	2010	592
	MADlibr	2017	83

issues, once new instances of this information may emerge later on in the bug repair process. The summation attributes type is intrinsically related to the report lifetime as the number of comments, watchers, and attachments grow as the bug report lifetime pass. The dataset dynamic perspective represents these moments when the report changes, when new information is added to the report or a field change, such as status change; priority definition or change; the writing of a new comment; a new collaborator starts to watch an issue.

The dynamic dataset is composed of three files:

1. `changelog`: This file stores every modification that ever happened on every Jira report field.
2. `comment-log`: This file stores information about each comment related to its report, and it was also mined from Jira.
3. `commit-log`: Some bug reports are related to some commit that fixes that bug. This file stores commit information about each report that has one. The file entries bring detailed information of each file modified by bug-fix commits and were mined from Github.

All the specific fields of each file can be found in Appendix A.

2.4 Dataset Characterization

This section presents a series of analyses and visualizations to characterize the dataset. The main goal is to provide a general picture of the bug fixing process based on the mined projects. Also, the provided analysis will serve as references in future chapters of this thesis to ground some decisions, approaches, and ideas.

2.4.1 Bug-Fixing Time

We define Bug-Fixing Time (BFT) the interval between the `CreationDate` and the `ResolutionDate`, *i.e.*, `BFT` = `ResolutionDate` - `CreationDate`. Foremost, we want to clarify that we use 'Bug Report Resolution Time', 'Bug fix effort' and 'Bug-fixing time' as synonymous in this thesis. In this manuscript context, they mean the same thing: the period when the bug was first reported until it is finally fixed. However, we know that the 'Report Resolution Time' is more faithful with this definition, but 'Bug-fixing time' feels more natural and usual in some contexts. Thus, we intercalate the terms depending on the context and avoid repetition.

Table 2 shows the frequency distribution for eight BFT intervals. The interval used

to group the reports by their resolution time is based on Saha *et al.* (2015). We choose to group the projects by category not individually for space concerns. The columns labels meaning are: ‘h’ (hour), ‘d’ (day), and ‘M’ (month).

Table 2 – The interval BFT distribution by category. We present the report’s BFT values in seven value range groups.

Category	BFT ≤ 1h	1h < BFT ≤ 1d	1d < BFT ≤ 7d	7d < BFT ≤ 1M	1M < BFT ≤ 6M	6M < BFT ≤ 12M	BFT ≥ 12M
big-data	1112 (4.39%)	4543 (17.96%)	7258 (28.69%)	5733 (22.66%)	4640 (18.34%)	929 (3.67%)	1087 (4.3%)
database	1264 (5.06%)	5250 (21.01%)	7611 (30.46%)	5292 (21.18%)	3893 (15.58%)	869 (3.48%)	806 (3.23%)
cloud	344 (5.24%)	1040 (15.85%)	1448 (22.07%)	1397 (21.29%)	1366 (20.82%)	381 (5.81%)	585 (8.92%)
web-framework	682 (12.28%)	1063 (19.14%)	1078 (19.41%)	866 (15.59%)	1087 (19.57%)	312 (5.62%)	467 (8.41%)
network-server	1172 (25.85%)	1317 (29.05%)	953 (21.02%)	496 (10.94%)	396 (8.74%)	98 (2.16%)	101 (2.23%)
library	422 (14.31%)	657 (22.28%)	589 (19.97%)	441 (14.95%)	487 (16.51%)	150 (5.09%)	203 (6.88%)
build-management	164 (12.28%)	263 (19.7%)	220 (16.48%)	214 (16.03%)	240 (17.98%)	74 (5.54%)	160 (11.99%)
machine-learning	114 (9.79%)	305 (26.2%)	242 (20.79%)	195 (16.75%)	245 (21.05%)	44 (3.78%)	19 (1.63%)
security	220 (24.2%)	265 (29.15%)	204 (22.44%)	121 (13.31%)	64 (7.04%)	27 (2.97%)	8 (0.88%)
All Reports	**5494 (7.5%)**	**14703 (20.06%)**	**19603 (26.75%)**	**14755 (20.13%)**	**12418 (16.94%)**	**2884 (3.93%)**	**3436 (4.69%)**

Based on Table 2, it is possible to see that the bug fixing time behavior is different given a software category, but the majority of bugs (83.88%) are fixed between 1 hour and six months. However, there are a few discrepant things, as 24.2% and 25.85% of security and network-server bugs, respectively, are fixed in less than an hour, while in categories like big-data, database and cloud, this value is between 4.39% and 5.24%. This discrepancy also appears on the BFT ≥ 12M, where the value for the build-management category is 11.99%, while in security is less than 1%.

Figs. 2 and 3 show the boxplot of the log-transformed (to deal with the skewed data) bug fixing-time. The reports with effort lower than one hour had their values truncated to ‘1’. The figures show, once more, that the effort resolution behavior of the reports is different among categories and projects.

2.4.2 *Priority*

The Jira ITS provides five default values for priority: *trivial*, *minor*, *major*, *critical*, and *blocker* (by order of priority). Only one project (Cassandra) does not uses the default values and uses three levels of priority: *low*, *normal*, and *urgent*. Figure 4 uses all projects (except Cassandra) to present some priority information. The visualization on the left shows the absolute number of the report with each priority. The projects have a different number of bug reports and distinct lifespans. For example, some of them have more than 16 years of development (*e.g.*, Hadoop and HBase), while others have less than six years of development (*e.g.*, SystemML and MADlibr). Thus, looking only at the absolute values could present a biased behavior from the

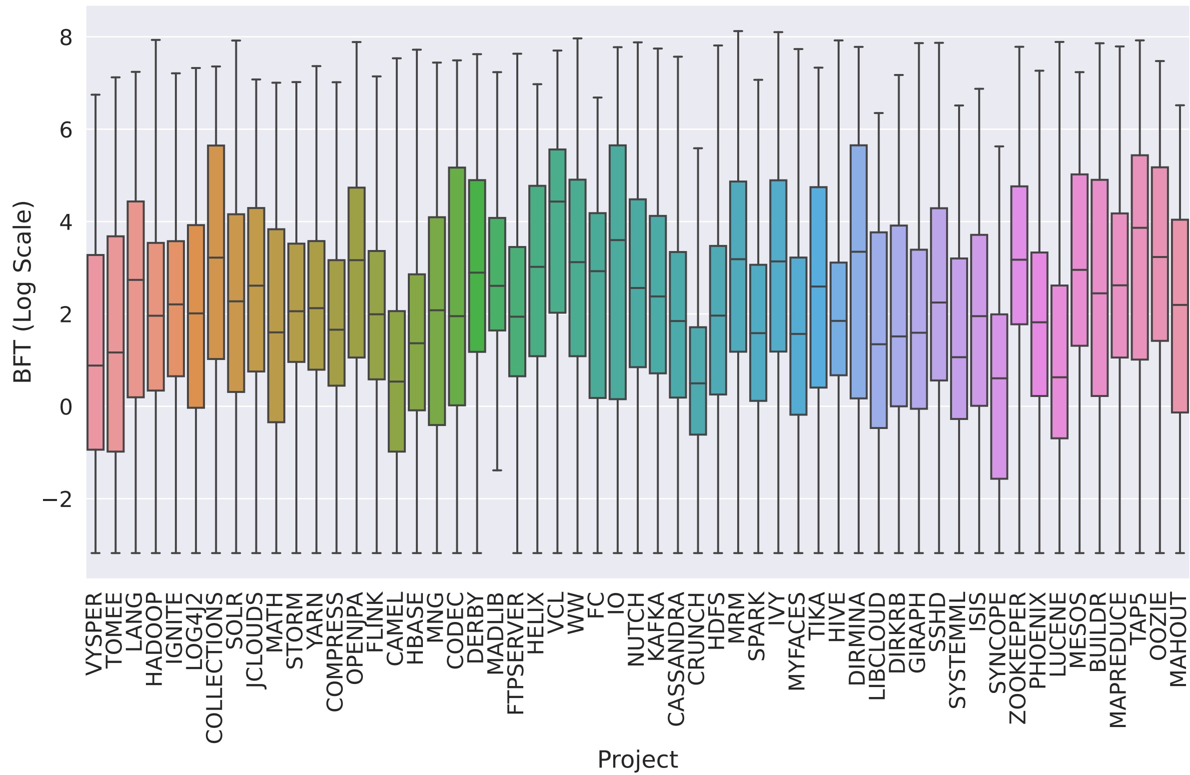


Figure 2 – Log-transformed bug-fixing time boxplot by project.

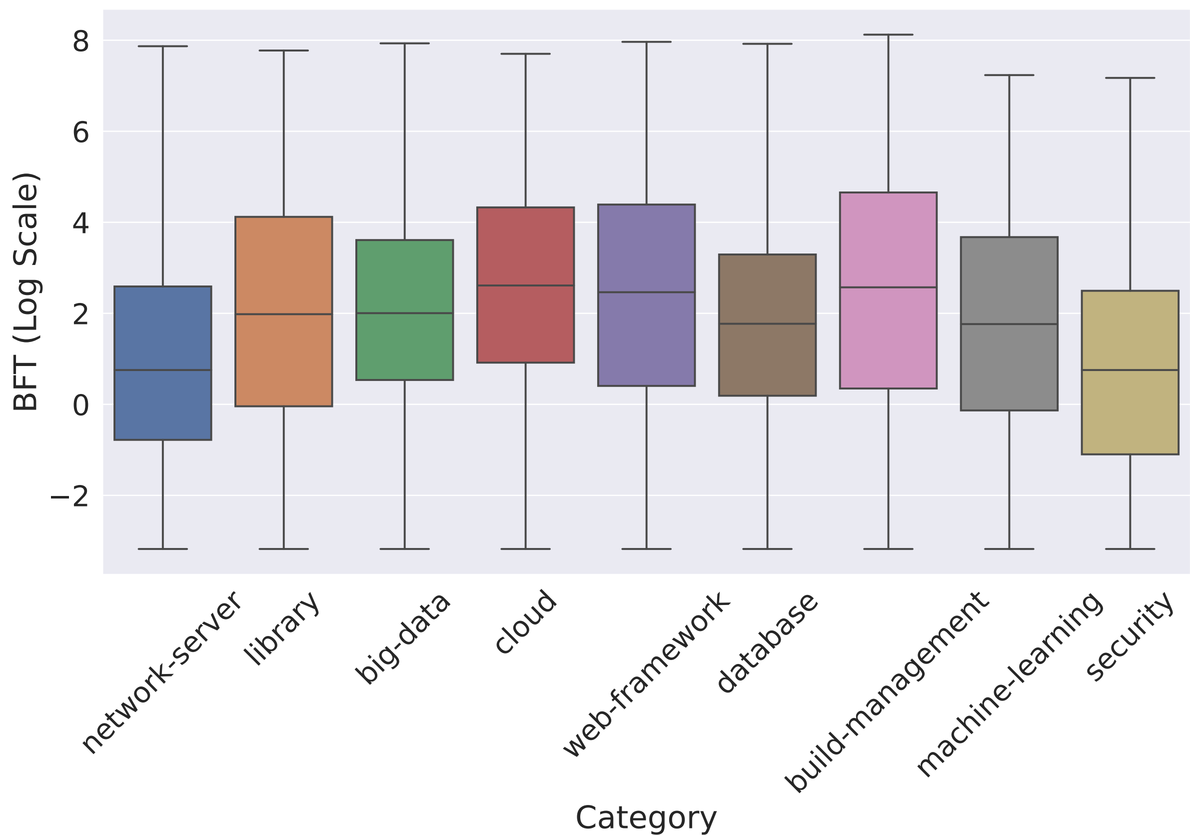


Figure 3 – Log-transformed bug-fixing time boxplot by category.

older projects. We create another visualization displayed on the right. The visualization shows the average proportion for each priority of all projects. The vertical line presents the standard deviation.

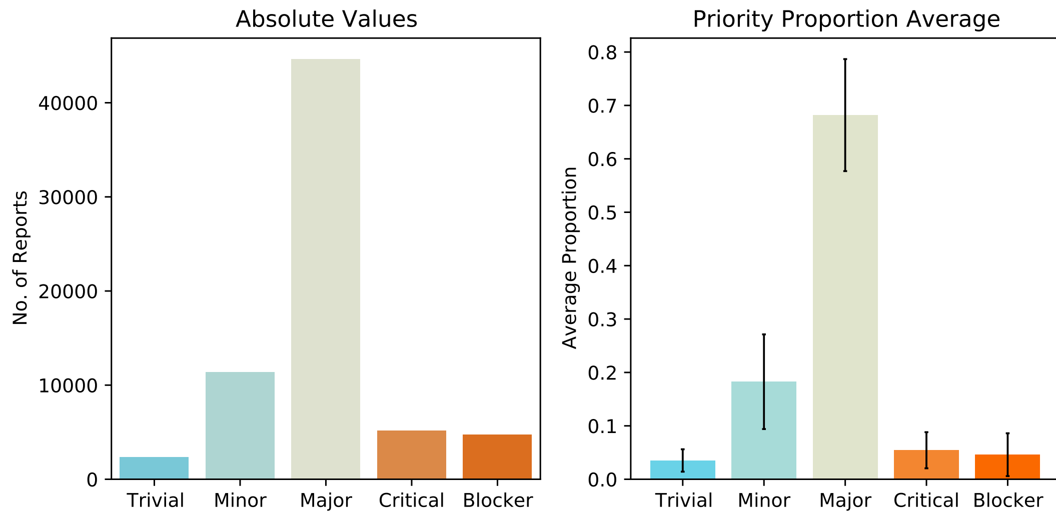


Figure 4 – Priority distribution. The absolute number of bug reports by priority across all the projects is on the left. The right visualization shows the average proportion value by priority across all projects.

The majority of reports present the *minor* or *major* priority. It is important to notice that *major* is the default value priority when a report is created, which may explain the high number of reports with this priority. The visualization on the right presents a similar behaviour when compared to the one on the left, but shows that the *trivial*, *critical*, and *blocker* can present smaller values, close to 0% in some projects.

2.4.3 Links

Two of the 53 features available in the `snapshot.csv` file are related to links between reports: 'InwardIssueLinks' and 'OutwardIssueLinks'. As the names suggest, given a bug report r_i with an 'Inward Link' reference to another bug report r_j, this indicates that somehow, r_j relates to r_i. Similarly, given a bug report r_i with an 'Outward Link' reference to another bug report r_j, this indicates that somehow, r_i relates to r_j. These concepts are easy to understand if we model the bug reports relations as an oriented graph: bug reports as vertices and links as oriented edges.

The *link-related* fields are string fields in the `snapshot.csv` file. If it is empty (NaN or 0), it indicates no link of the specific field (Inward or Outward) type. If not empty, the field contains a string of unique Keys separated by a line break (if there is more than one link). The

keys in the Jira ITS follow the format {project}-{number}. Hence, given the project 'Spark', examples of keys would be SPARK-213 and SPARK-481. It is important to notice that a project report can reference another project through an issue link. Different types of issues (bugs or enhancements, for instance) can reference another type of issue.

Fig. 5 explores the possible scenarios around links between bug reports. It shows seven bug reports of the Spark project in five columns, in order: (i) the Pandas Dataframe[12] index; (ii) the bug report unique key; (iii) the Inward links references; (iv) the Outward links references; (v) the total report number of links.

	Key	InwardIssueLinks	OutwardIssueLinks	TotalLinks
1	SPARK-585	0	0	0
42	SPARK-694	0	Reference:SPARK-668	1
208	SPARK-1190	Duplicate:SPARK-1067	Reference:SPARK-3782	2
212	SPARK-1199	Duplicate:SPARK-1836\nDuplicate:SPARK-2330\nRe...	Reference:SPARK-2452\nRegression:SPARK-2576	6
312	SPARK-1493	0	Reference:CALCITE-746	1
425	SPARK-1828	0	Duplicate:SPARK-1802\nDuplicate:HIVE-5733	2
422	SPARK-1825	Duplicate:SPARK-5164\nReference:YARN-2929	0	2

Figure 5 – Example of links using bug reports from Spark project. There are several scenarios and types of links in bug reports.

We explore each scenario of bug report regarding how it relates with another report, using the examples presented in Fig. 5, as follows:

- The bug report 'SPARK-585' is a bug report with no association with another report. This report is an example that represents the **first scenario**, *reports with no links*.
- The others six reports represent different cases of reports with links. The bug reports 'SPARK-1190' and 'SPARK-1199' are two examples of reports with references in both link fields. Both examples show that a report can have more than one reference in the same field. These reports are examples of the **second scenario**, *reports with both links types*.
- The report 'SPARK-1825' is a report with links only in the 'Inward' field. Notice that it has two references, an internal reference (to a report in the same project, 'SPARK-5164') and an external reference (to another project, Yarn, 'YARN-2929'). This report is an example of the **third scenario**, *reports with references only in the 'Inward' field*.
- The 'SPARK-694', 'SPARK-1493', and 'SPARK-1828' are reports with links references only in the 'Outward' field. The 'SPARK-694' references an internal report, while the

[12] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

‘SPARK-1493’ has an external reference to the project ‘Calcite’. The ‘SPARK-1828’ has two references for others issue reports, one internal and another external, ‘Hive’. These are examples of the **fourth scenario**, *reports with references only in the ‘outward’ field*;

– We can also explore these relations by looking only at the reports with links. As shown in the examples, a report may reference an external project. We can split the reports with links references into two groups: *reports with only internal references and reports with external references*. This is a **fifth scenario** we also explore in the analysis.

Fig. 5 also shows that each link is associated with a word that describes its nature. The examples are "Reference", "Duplicate" and "Regression".

Fig. 6 shows the proportion of different links related scenarios of bug reports, as explained earlier. The visualization is a horizontal stacked bar and presents the proportion of different link-related scenarios of bug reports.

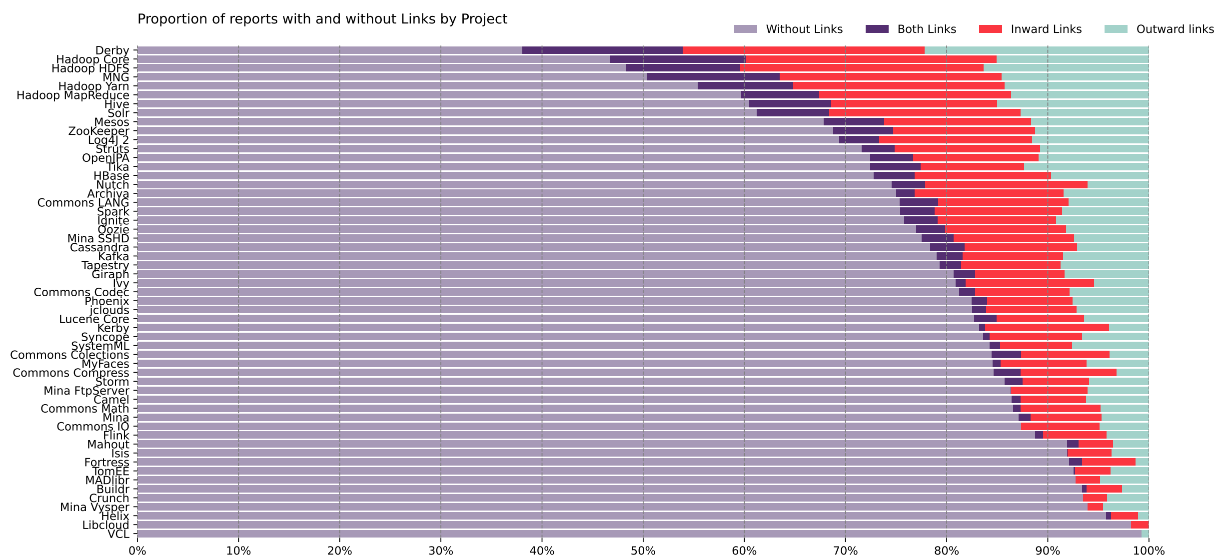


Figure 6 – The proportion of different scenarios of links between bug reports. Most projects have more reports with no link than reports with links.

We highlight a few perceptions:

– In all projects, the proportion changes significantly. For example, the ratio of reports with some link reference varies from more than 60% in Derby down to almost none in VCL.

– Only three projects (Derby, Hadoop Core, and Hadoop HDFS) have more reports with links than reports with no links.

– The occurrence of reports with both links references occurs less frequently than those with only one link.

– We also notice that reports with inward references are slightly more usual than reports

with outward references for most projects.

2.4.4 The Changelog Dataset

The changelog dataset contains 853,190 entries regarding 63 different types of changes. The top 15 most common changes are listed in Table 3. The first column is the field where the change happened. The second column is the number of changes in the specific type and represents the total changes percentile. The third column is the number of unique bug reports where the related change type happened since the same type of change can occur in the same report more than once.

Table 3 – The top 15 most common fields updates in bug reports.

Changed Field	**Number of Changes**	**Unique Bugs**
Status	189552 (22.21%)	73273 (100%)
Fix version	140758 (16.49%)	58255 (79.48%)
Attachment	114998 (13.47%)	41012 (55.95%)
Resolution	84347 (9.88%)	73273 (100%)
Assignee	56077 (6.57%)	40040 (54.63%)
Remoteissuelink	41525 (4.86%)	12827 (17.50%)
Workflow	36766 (4.30%)	23674 (32.30%)
Link	36464 (4.27%)	20843 (28.43%)
Description	21367 (2.50%)	13315 (18.16%)
Version	21232 (2.48%)	10577 (14.43%)
Component	13936 (1.63%)	10119 (13.80%)
Hadoop flags	13887 (1.62%)	13626 (18.59%)
Summary	13003 (1.52%)	10485 (14.30%)
Priority	12317 (1.44%)	10924 (14.90%)
Target version/s	7828 (0.91%)	5237 (7.14%)
Total	**417,480 (48.93%)**	–

A few things are worth being highlighted as we analyze Table 3. First of all, it is natural that every report has Status and Resolution changes. All of them starts with OPEN status and concludes with a CLOSED or RESOLVED status; The same thing with the Resolution field: it starts with a *NaN* resolution and concludes with a *Fixed* resolution. It makes sense that the Status field is where the higher number of changes occurs. The bug report has several intermediary status states until it gets a CLOSED or RESOLVED status. The Assignee field changes are the fifth on the list and are present on almost 55% of the report. It is worth investigating this because the Assignee is a fundamental piece in the bug-fixing process. One can argue that the bug-fixing process *per si* starts with the Assignee definition or when there is some kind of

response (a comment, a change of a field, etc.) after the bug report creation, once it is hard to believe that the bug will be fixed if no one is responsible for it. A little more than half of the reports have some `Assignee` change, indicating that one is defined when the report is created and never changes. Another scenario is that the bug is fixed by a person that creates the report and has never been set as `Assignee` on the Jira report. The bug can be identified as duplicated or even that it is not a bug before the `Assignee` definition step. The last thing to highlight is the priority changes. Priority is an object of study on several bug report related papers (TIAN *et al.*, 2013; TIAN *et al.*, 2015; SHARMA *et al.*, 2012; UMER *et al.*, 2018). The relatively low number of changes (only on 14.90% of the reports) can be evidence that the initial hint, defined on the bug report creation, is precise.

2.4.5 Reports Updates

We define a *report update* as the change, addition, or deletion of a report field or comment. Examples are the creation of a new comment, the change of the report status, priority, or any other report attribute. Fig 7 shows the distribution of the number of updates, considering every bug report in the mined dataset. The report updates play a significant role in the next chapter, so it is necessary to represent more details.

The minimum and maximum values are, respectively, 2 and 595 updates. Other descriptive statistics are the mode (10), median (16), and the third quartile (25).

2.4.6 Status Changes

The status changes define which phase in the report life cycle a bug report is. As we can see in Fig. 1, there is a set of standard Jira statuses, but new ones can be created for a customizable workflow. Table 4 lists all the status values that appear on the dataset `Status` field. The first column is the status fields names. The second and third ones are the numbers of occurrences of a given status in the `From` and `To` fields of the dynamic perspective dataset (`changelog`), respectively. This means the number of status changes that had originally the specific status (`From`) and the number of status changes that changed to the specific status (`To`). The fourth column is the number of projects that have at least one report that had the given status in its lifetime. We also verified six exclusive categories related to two specific projects.

Table 4 shows that the “Open”, “Resolved”, “Closed” and “Reopened” are the most frequent status values. They are present in all projects of the dataset, being a “From” or a “To”

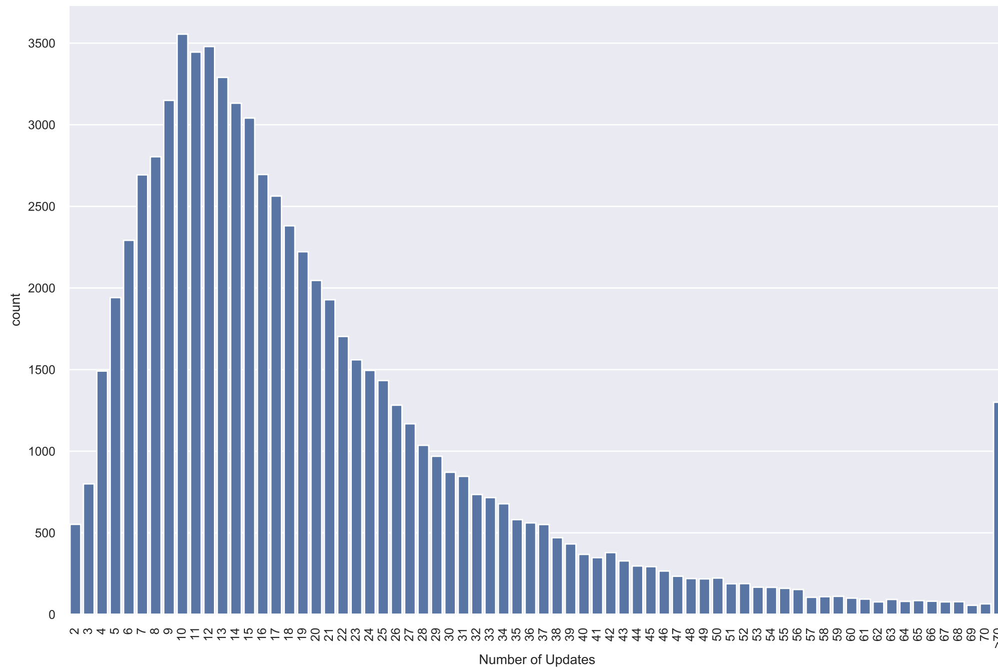


Figure 7 – The number of reports updates histogram. The mode is ten updates and the median sixteen updates. Most reports have between four and twenty-five updates.

Table 4 – Bug reports possible 'status' values. Jira provides a set of standard values, but it is possible to create customs for a specific project (cases of Mesos and Cassandra).

Status	**No. on** `From`	**No. on** `To`	**No. of Projects**
Open	84562 (55)	11280 (40)	All Projects
Resolved	36027 (55)	72766 (55)	All Projects
Patch Available	41842 (17)	41839 (17)	17 Projects
In Progress	15669 (51)	15669 (51)	51 Projects
Reopened	5465 (55)	5463 (55)	All Projects
Closed	1681 (43)	38229 (55)	All Projects
Accepted	1998	1998	Mesos Only
Reviewable	1535	1534	Mesos Only
Ready To Review	32	33	Mesos Only
Ready to Commit	611	611	Cassandra Only
Awaiting Feedback	81	81	Cassandra Only
Testing	49	49	Cassandra Only

value. The “MESOS” and “CASSANDRA” projects have their own exclusive `Status` values. This probably has the same explanation of ”CASSANDRA” priority unique values: these are exclusive project status values defined by the project company’s bug triage process.

Let $S = \{S_1$ = “Open”, S_2 = “Resolved”, S_3 = “Patch Available”, S_4 = “In Progress”, S_5 = “Reopened”, S_6 = “Closed”$\}$ be a set of status changes of all projects. Table 5 shows the adjacent matrix of status changes. We remove the exclusive status values occurrences of “Mesos” and “Cassandra” projects for a more general scenario.

Table 5 – Adjacent matrix of status changes.

		To					
		S_1	S_2	S_3	S_4	S_5	S_6
From	S_1	↗	↗	↗	↗	↗	↗
	S_2					↗	↗
	S_3	↗	↗	↗	↗	↗	↗
	S_4	↗	↗	↗			↗
	S_5		↗	↗	↗	↗	↗
	S_6		↗			↗	

An analysis of Table 5 reveals the common behavior of a bug report.

- From “Open” (S_1) status, there are changes to all status (include “Open” and “Reopened”), which is natural since “Open” can be seen as an initial report state. It goes in its majority to “Patch Available” (42.58%) and to “Resolved” (36.11%), occasionally to “In Progress” (14.71%) and “Closed” (6.53%).
- From “Resolved” (S_2) status, occasionally goes to ”Reopened” (9.55%) and generally to ”Closed” (90.45%) status.
- From “Patch Available” (S_3), it generally goes to “Resolved” (70.20%) (which may imply that the Available Patch is correct) and occasionally goes to “Open” (26.27%) or “In Progress” (3.43%) status (which may imply that the Available Patch was not correct / accepted). There is a tiny number of changes to “Patch Available”, “Reopened” and “Closed”.
- From “In Progress” (S_4) status, it goes in its majority to “Resolved” (57.96%) or “Patch Available” (32.11%) status and occasionally to “Open” (4.32%) or “Closed” (5.11%) status.
- From “Reopened” (S_5) status, it goes in its majority (63.49%) to “Resolved” status, occasionally to “Patch Available” (17.41%), “Closed” (13.29%) or “In Progress” (5.16%)

status.

– From “Closed” (S_6) status, almost every change (99.997%) is to “Reopened” status, with a tiny number (0.003%) of changes to “Resolved” status.

Next, we verify the distribution of status changes on the reports. Fig. 8 shows this information. We remove the status changes that register the same status for this visualization. As shown in Table 5 there is a number of redundant status *i.e.* there is the register of status change, but the final status is equal to the original status. We use all the projects in this visualization to see the number of status changes, not the real state status. Most reports (90.03%) have between 1 to 4 status changes.

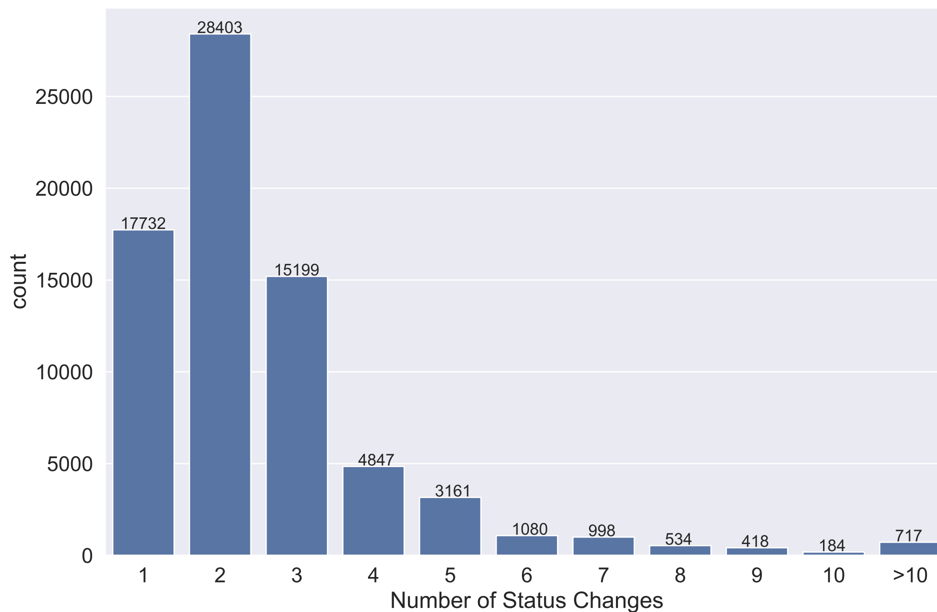


Figure 8 – The number of status changes by reports. The great majority of reports have between two and four status updates.

2.5 Dataset Relevance

Our dataset provides information about the bug-fix process documentation on ITS. We believe that the dataset relevance can be justified by a few factors we describe in the following.

The first factor is the number of projects (55 projects), its different categories (9 categories), and release dates (2003-2017). This diversity can lead researchers to find specific to general differences in the bug-fix process. Several works use ITS reports to understand

the bug-fix process better. For example, our proposed dataset can be used to replicate those studies and confirm or not their evidence once a number of them use fewer projects to assess their hypothesis (CANFORA *et al.*, 2011; ZHANG *et al.*, 2012; HABAYEB *et al.*, 2018; AKBARINASAJI *et al.*, 2018). The mining process is not a trivial task, and making the dataset available is a shortcut for all the researchers who want to better understand the triage and bug-fix process.

The second factor is its dynamic aspect (i.e., the changes that occurred during the bug-fix activity). Other works (HABAYEB *et al.*, 2015; XU; ZHOU, 2018) highlight the report changes relevance once it is not wise to perform a conclusive analysis only on a set of reports snapshot. However, since our dataset has all the changes and new additions on a report, more detailed and reliable conclusions can be made.

2.6 Related Work

Several papers propose datasets composed of ITS reports. This section discusses the works that have a similar approach or goal with our proposal.

Lamkanfi *et al.* (2013) describe the mining process to collect bug reports of Eclipse and Mozilla issues tracking systems. They organized a total of 46,884 examples from 4 Eclipse products (Eclipse Platform, JDT, CDT, and PDE) and 168,024 from 4 Mozilla products (Mozilla Core, Mozilla Firefox, Mozilla Thunderbird, and Mozilla Bugzilla). The authors set the dataset focus on the updates or changes of the reports.

Habayeb *et al.* (2015) propose a Firefox Temporal Defect Dataset with 86,444 bug reports covering eight years from 2006 to 2014. The work points that generally, researchers have focused "mostly on analyzing the frequency of the occurrence of defects and their attributes". This simplistic analysis does not consider the temporal alignment of events that leads to those changes. Instead, based on a performed exploratory analysis on the dataset, the authors suggest that their mined data can be used to predict defect proneness or defect sizing and suggest potential defect fixes as well.

The work presented by Ortu *et al.* (2015) is probably the most similar to our proposal. The paper presents a dataset composed of more than 1K projects with more than 700K issues reports and more than 2 million comments from the issues. All issues are extracted from the Jira ITS of four open-source ecosystems: Apache Software Foundation, Spring, JBoss, and CodeHaus. They argue that using the amount of information regarding comments can lead to

sentimental and technical analysis. The major point that differs their database from ours is that they mined all issues (not only bugs as ours). Their dataset does not provide information regarding source configuration management and does not provide a dynamic perspective of the reports.

The work developed by Zhu *et al.* (2016) describes the dataset mined over Mozilla Issue Tracking History. The dataset contains more than 774,000 issues reported from April 1998 to January 2013, and it is composed of several issues types: defects, feature requests, and enhancements, among others. The particular thing on this proposal is that it includes multi-extract (they retrieve data from two channels, a front-end, and back-end, at three different times); and a multi-level design (Level 0 is the raw data and Level 1 is an after processing standardized data).

Xu and Zhou (XU; ZHOU, 2018) present the Multi-level dataset mining process of the Linux Kernel Patchwork is shown. The dataset is composed of 665,550 patches information extracted between December 2008 and December 2017 from the Linux Kernel Mailing list. They organize the data on three levels: Level 0, the raw data; Level 1 as structure data; and Level 2, as a further processing result. The authors emphasize reducing researchers' effort in collecting, cleaning, and processing after initiatives like these.

3 THE ROLE OF BUG REPORT EVOLUTION IN RELIABLE FIXING ESTIMATION

In this chapter, we describe the process to propose and evaluate a new approach to bug-fixing time estimation models. Given the scenario composed of the relevance of bug report resolution estimation and the changeable and evolutionary nature of bug reports presented in the Chapter 1, we investigate three main questions in this chapter. We formalize our research questions below:

- **RQ1: How frequently are the bug reports' fields updated, and how do these updates impact models for fixing time estimation?** To answer this question, we first analyze the most common reports' fields updates of ten open-source projects. Next, we replicate the seminal work by (ZHANG *et al.*, 2013) (more details about why we select it on Section 3.1.2) using reports in different stages of their evolution. This way, we can verify if the estimation models present any performance variation when we consider information from the various possible states of the bug reports.
- **RQ2: What is the most promising model configuration to build reliable models for fixing time estimation considering bug reports at different stages of evolution?** To answer this question, we evaluate three different machine learning models trained with data from ten Apache software projects. We look at the fixing time estimation capability as a combination of two perspectives: i) data balancing strategies *ii*) the estimated label related to the resolution time. To achieve that, we created a temporal dataset based on our presented dataset in Chapter 2 and evaluate the estimation models with several metrics.
- **RQ3: To what extent is there a moment in the bug report life cycle where a resolution estimation is more precise?** We already discussed (and will further detail in Section 3.1) the idea of bug reports evolution in Chapter 2. To tackle this question, we look at every report state as an individual and independent report. After training the models with such approach, we identify when in the report life cycle we obtain the best estimates. We perform a posterior analysis and get a sense of the difference in the accuracy of the estimations at the different steps of the bug life cycle.

3.1 Materials and Methods

This section describes the materials and methods used to address our research questions. Subsection 3.1.1 explains the dataset used in all the investigation steps, created

based on the dataset presented in Chapter 2. In the Subsection 3.1.2, we present the necessary information to answer RQ1. Subsections 3.1.3 and 3.1.4 present the materials and methods to answer, respectivaly RQ2 and RQ3. We conclude this section describing in subsection 3.1.5 the process to create the train/test data partition to train the machine learning models.

3.1.1 A Temporal Dataset of Bug-Fixing Activities and Reports

Using the dataset proposed in Chapter 2, we intend to use the bug report information from the Jira issue tracking system to answer the research questions. There are several independent variables available in the reports, and they can be used to train machine learning models. All the attributes are described and explained in Appendix A, and the subset that we use in our experiments are listed in Section 3.1, Table 9.

Most related work in the literature have been dealing with bug fixing time estimation as a type of effort prediction. The usual effort to be estimated is *time* itself, but there are other choices, such as person-hour or code churn. Therefore, the main idea is to model the task as the necessary effort to fix the bug, i.e., the resource applied to change the code or remove fractions of code that lead to bugs.

In the current work, we look to model the effort estimation as the bug report resolution time. Thereby, we have as a dependent variable a derivative attribute from two of the report's fields. Thus, we define the Report Resolution Time (RRT) as the difference between the *report resolution date* and the *report creation date.*

Definition 1 *Let* ***Creation Date (CD)*** *be the report's Creation Date and let* ***Resolution Date (RD)*** *be the report's Resolution Date. The variable* ***RRT****, i.e. the Report Resolution Time, is defined as* ***RRT = RD - CD****.*

The dataset original structure proposed in Chapter 2, as it is, limits the potential of applicability and confidence of possible RRT estimations. It is crucial to notice that the `snapshot` files contain the bug reports in their final state, i.e., the values of their fields at the moment when they are closed. The idea is that if one uses the dataset as presented to estimate RRT, it may lead to optimistic estimations because the snapshot file contains information of the bug report last state. Hence, the report features values may contain information not available when performing the report triage in its initial state. For the rest of this Chapter, the state of a bug report will be discussed more often. Hence, we formally define the state of a bug report below.

Definition 2 *A bug report's **state** is comprised of its attributes' set values in a given moment of the life cycle, right after one or more of its attributes are updated (deleted, changed, or added). The report **initial state** is the set of its attributes' values right after its creation. The report's **final state** is the set of its attributes' values right after the report is resolved/closed. The report's **intermediary states** are the states between the **initial** and **final** state.*

Since the `snapshot` files only contain the final state of each report, it is not logical to build RRT predictive models using only the reports' final state, once they only provide data related to the report's resolution. For instance, the number of comments and their top words are cumulative attributes that change and increase during discussions made by the developers. It would be desirable to have a management tool that estimates the effort for intermediate reports' states to ensure that for each state exists an associated **RRT**.

The complete dataset does not contain all states of each report. However, it provides the necessary information to obtain them. Every state of each report can be re-created from the three other files of the dataset: `snapshot`, `changelog` and `commentslog`. We wrote a Python script that, for each report in the `snapshot` file, re-creates the report's state for every change and update that ever happens in the report's life cycle. Fig. 9 summarizes such a re-creation process. The correspondent pseudo-code is presented in Algorithm 1 and described as follows. In **Line 1**, we define a structure to store all the reports' states re-created by the script. In **Line 2**, we get each final report state at a given time from the `snapshot.csv` file to re-create its previous states. In **Line 3**, we include the selected report state into the `temporal_dataset` structure since it is the final state report. In **Lines 4 and 5**, we get all the changes and comments corresponding to the current report. In **Line 6**, we group the report fields' changes and comments additions that co-occur or occur with a small difference in time (5 seconds) in the same structure and call it an update. Each update is what defines the difference between two states. In **Line 7**, we order the updates by date and time in a descending way, so the last updates are on the top of the structure. Hence, we are re-creating the states in decreasing order (from the final state report to the initial state report). We start the process by using the final report state s_n (given at **Line 2**), where n represents the number of states that the report has. Next, we re-create the previous report states down to the initial state $(s_{n-1}, s_{n-2}, ..., s_1)$. In **Line 8**, we perform a simple attribution to set the current state to be used to re-create the previous state. In **Line 9**, we go through each report update to re-create the previous report states. In **Line 10**, a new report state is re-created. A new report state s_{i-1}, is re-created using the current state report s_i and the current update variable

from **Line 9**. This method undoes the updates (registered in the update variable) that made the report goes from its state s_{i-1} to its state s_i. In **Line 11**, we include the recently re-created report state in the `temporal_dataset`. In **Line 12**, we update the last created report state to be used to build the previous report state. In **Line 15**, we add a new column RRT to the dataset, which is calculated as detailed in Definitions 1 and 3.

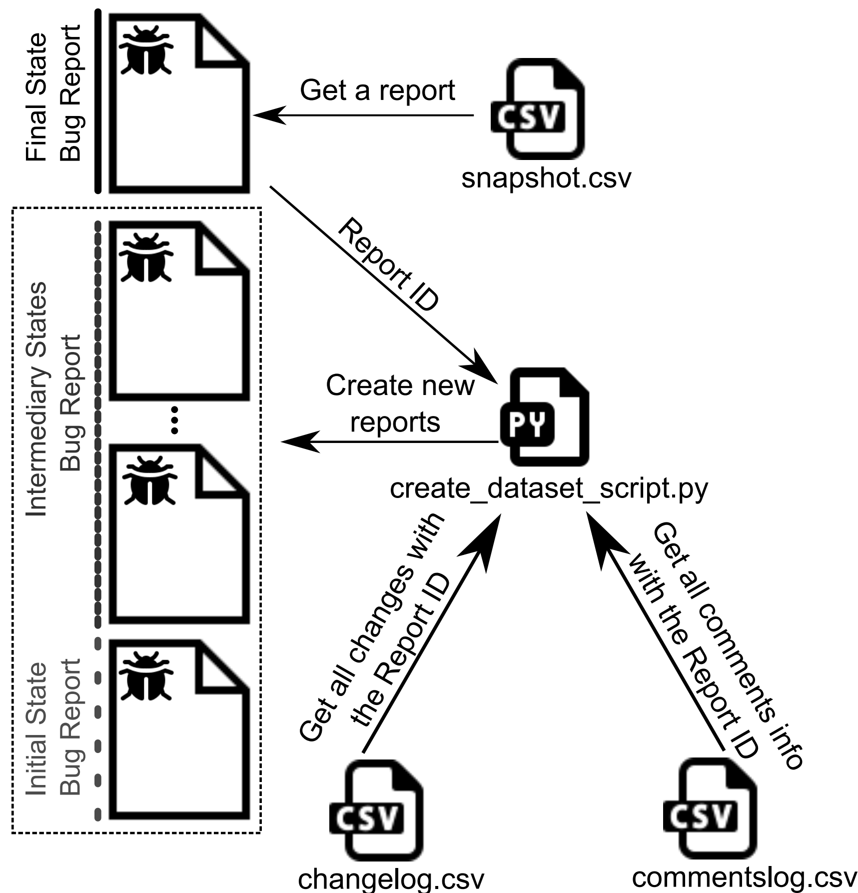


Figure 9 – The temporal dataset of bug-fixing activities and the reports states re-creation process.

Here, we want to highlight a crucial aspect to understand our approach. In the `temporal_dataset`, we have several bug reports in different states. From now on, to train the machine learning models, **we will consider every report, regardless of its state, as an independent report**. When we create the `temporal_dataset`, we can use every report (initial, intermediate, or final report states) as individual patterns to train machine learning models. We argue that if the actual report field values are enough to predict when it will be closed/resolved, the models will provide different estimations with different reports' states. Every report state has its attributes (fields) values and an RRT value associated. Thus, we expand the idea of RRT to each report state as follows, already using the idea of a report state as an independent report.

Algorithm 1 Temporal Reports Dataset Builder Script

```
Require: snapshot.csv,changelog.csv,commentlog.csv
 1: temporal_dataset = []
 2: for final_report_state in snapshot.csv do
 3:     temporal_dataset.append(final_report_state)
 4:     comments = get_comments_by_key(final_report_state.key) #from commentlog.csv
 5:     changes = get_changes_by_key(final_report_state.key) #from changelog.csv
 6:     group_of_updates = group_by_datetime(comments, changes)
 7:     group_of_updates = group_of_updates.order_by_datetime()
 8:     current_state = final_report_state
 9:     for update in group_of_updates do
10:         new_state = delta_state(current_state, updates)
11:         temporal_dataset.append(new_state)
12:         current_state = new_state
13:     end for
14: end for
15: temporal_dataset = calculates_RRT(temporal_dataset)
16: return temporal_dataset
```

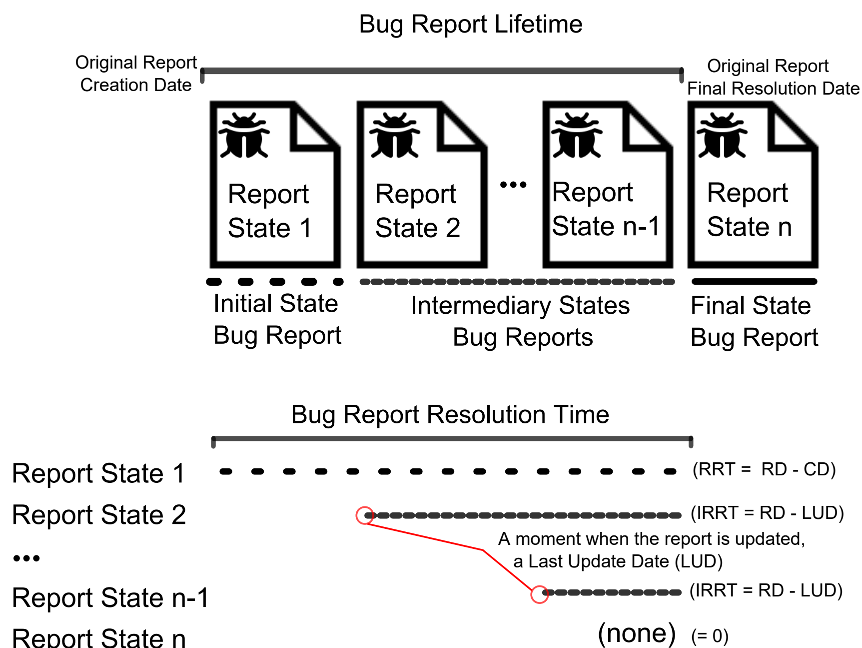


Figure 10 – Bug reports resolution time calculation. The RRT is calculated based on the current state of the report.

Definition 3 *The calculation of each report RRT depends on its state type (as seen in Definition 2):*

- *The **RRT** of an **Inital State Bug Report** is calculated as the **RRT** established on Definition 1.*
- *The **RRT** of an **Intermediate State Bug Report** is calculated as follows: let r_i be a **Intermediate State Bug Report**, with i indicading the state that this report represents. The existence of r_i implies that a previous report state $r_{(i-1)}$ exists, which can be another **Intermediate State Bug Report** or an **Inital State Bug Report**. For r_i to exist, a set of fields in report state $r_{(i-1)}$ was updated at some moment of the bug report life cycle. Let the Last Update Date (LUD) be the update moment. The RRT of an **Intermediate State Bug Report** is defined as **RRT = RD - LUD**.*
- *The **RRT** of a **Final State Bug Report** is defined as an **Intermediate State Bug Report**. However, it has a particularity: the **LUD** value represents the moment when the report is closed (i.e., there is a change of the status to closed/resolved). Thus, if LUD = RD, then RRT = 0.*

The above definition can be interpreted as a simple variation of Definition 1: every time the report is updated, a new report (state) is created. Hence, the LUD can be seen as the CD of a new state report. Another way to look at the RRT calculation is that a bug report RRT is the time that will take to resolve the current report state. Fig. 10 summarizes the definition.

3.1.2 Bug Reports' Fields Updates and Zhang et al. (2013)'s Work Replication

For the bug reports' fields updates analysis, we use the `changelog.csv` file of each project to list the fifteen most common fields updates. We also verify how often previously proposed approaches to the report resolution estimation task use those attributes. We select the following related work: Zhang *et al.* (2013), Assar *et al.* (2016), Al-Zubaidi *et al.* (2017) and Habayeb *et al.* (2018). Those and other related papers that deal with bug-fixing time estimation are summarized in Section 3.5. The approaches mentioned above are also candidates to be compared with our approach, using temporal dataset. The main reason to select them are: (i) the similarity with our approach and (ii) the impact factors of their publication site. However, three of them present some shortcomings when analyzed. Assar *et al.* (2016) conclude that their approach does not present good results and is not applicable. Al-Zubaidi *et al.* (2017) models the problem as a regression task, while we model it as a classification task. The work by Habayeb *et*

al. (2018) uses several attributes that we cannot calculate due to dataset differences. Hence, we only consider the work by Zhang *et al.* (2013) as a comparison baseline. Nevertheless, we still consider the other three works when analyzing the field updates.

The work developed by Zhang *et al.* (2013) uses a K-Nearest Neighbors (KNN) algorithm with a set of standard report's attributes. Table 6 lists their names, descriptions, and if it is present or not in the dataset proposed in our dataset.

Table 6 – Attributes used by Zhang *et al.* (2013).

Attribute Name	Description	Present or Equivalent on Our Dataset
`Submitter`	The bug report submitter.	Equivalent to `Reporter`
`Owner`	The developer who is responsible for resolving the bug	Equivalent to `Assignee`
`Severity`	The severity of a bug report	No
`Priority`	The priority of a bug report	Yes
`ESC`	Indicator of whether the bug is reported by end users or by the QA team.	No
`Category`	The category of the problem	No
`Summary`	A short description of the problem.	Yes

We only use a subset of the original attributes, as our dataset does not contain all of them, as shown in Table 6. This is a limitation of the Jira platform that does not provide these missing attributes by default. In the data pre-processing step, the Submitter and Owner are encoded using the one-hot encoding. The work by Zhang *et al.* (2013) uses the standard Euclidean distance for most of the attributes in the KNN algorithm, except for the Priority, Severity, and Summary, which have specific representations. The Jira platform, by default, considers five levels of priority: Trivial, Minor, Major, Critical, and Blocker. They are encoded in an ordinal way, with numbers from 1 to 5. The Summary field is a set of words after the removal of stopwords. Hence, based on the baseline work, the functions used to compute the difference between two priorities, $d_p(P_a, P_b)$, and two summaries, $d_s(S_a, S_b)$, are given by

$$d_p(P_a, P_b) = |P_a - P_b| \times 0.2. \tag{3.1}$$

$$d_s(S_a, S_b) = 1 - \frac{|S_a \cap S_b|}{|S_a \cup S_b|}. \tag{3.2}$$

Both Equations 3.1 and 3.2 are adaptations from the work by Zhang *et al.* (2013). Equation 3.1 was adjusted because the original paper's projects have four priority levels, while in the Jira platform, the reports have five. The function represents a weighted distance between two priorities (e.g. Trivial bug reports are closer to minor than to Blocker ones). In Equation 3.2, the original paper uses another set of words W_C in the equation. They represent *the set of standard words extracted from pre-defined category labels*, but the text does not detail how those words are selected. Hence, we chose to remove it as both proposals (the original and Equation 3.2) have the same idea to measure the text similarity.

There are four attributes used to train the models using the baseline approach, and we call them *Set 1*. We also define a second set of attributes, *Set 2*, composed by eleven attributes. This *Set 2* is similar to those we use in our approach. We use those new attributes to verify if they could improve the results. Concomitantly, it also provides a more fair way to compare our approach to the baseline. The Table 7 presents the list of attributes for each *Set*.

Table 7 – Description of the two 'sets' of attributes used in the baseline approach.

Set of Attributes	List of Features
Set 1	`Summary`, `Priority`, `Reporter`, and `Assignee`
Set 2	`Summary`, `Priority`, `Reporter`, `Assignee`, `Comments`, `Description`, `NoAffectsVersions`, `NoComponents`, `NoAttachments`, `TotalLinks`, and `NoAttachedPatches`

We draw three experiments for each set of attributes to test our hypotheses using the baseline approach (ZHANG *et al.*, 2013). **1)** In the first experiment, we train and test using only the final state reports (`EXP1`); **2)** in the second experiment, we train and test using only the initial state reports (`EXP2`); **3)** in third experiment, we train using final state reports and test with initial state reports (`EXP3`). In the first and second experiments, we intend to verify if the results are different depending on the state used to train the model. They also provide a baseline method to compare with our approach. For the third experiment, given that there is a difference depending on the state used to train the model, we want to verify if such a scenario is still applicable in practice (*i.e.* it does not matter to train using the last state reports as long as the models are able to estimate good results using the initial reports). It is worth noting that the RRT as established in Definition 3 is not used in this round of experiments. The RRT is independent of the state (initial or final), being the *actual* RRT as established in Definition 1. Also, to avoid ambiguity, we highlight that the three EXP use the same train/test splits and sizes. For each project, we first created a 5-fold partition using the reports' ID (e.g., we do not look at the report features at this point). For each report, through its unique ID, we recovered the values of the attributes to train and test the models depending on how each EXP is defined. We discuss and describe this process at length in subsection 3.1.5.

For each one of the ten projects dataset, we train the models for each experiment setting and calculate the average accuracy and f-measure by considering a 5-fold cross-validation. The work Zhang *et al.* (2013) uses the concept of a `time unit` to separate the reports between two classes. A project's `time unit` is the median of its report resolution time. The original approach presented by Zhang *et al.* (2013) tests five different thresholds to split the data: 0.1,

0.2, 0.4, 1, and 2-time units. For instance, consider a hypothetical project with a median report resolution time of seven days. If we train a model by splitting the data with the 0.1-time unit, the model predicts if the report will take more than 0.7 days (approximately 16,8 hours) to be resolved. Splitting the data with the 0.2-time unit predicts if the report will take more than 1.4 days (approximately 33,6 hours) to be resolved. When considering the 1-time unit, if a given report will take more than seven days to be resolved, and so on. We include in the list of thresholds to split the data in five and ten days thresholds. We do this because these are the values we use to split the data in our approach (more on why we choose these thresholds can be found in subsection 3.1.4). Thus, it will help us to compare both solutions (our approach and the baseline approach).

3.1.3 Preprocessing steps on the 'Temporal Dataset' to apply our approach

For our approach, we chose to use the temporal dataset process creation on 10 of the 55 projects from the original dataset, namely: Hadoop Core, Hadoop Yarn, Hadoop HDFS, Hadoop MapReduce, Lucene, Flink, Solr, Zookeeper, Kafka and Spark. The project selection criteria are project maturity (years of development) and the number of bug reports. After applying the previously described script to each of the project's datasets we have the data to train the machine learning models. A few aspects guide us to select these ten specific projects. Each dataset project to be used in our approach increases the time and computational power dramatically. Creating the temporal dataset of each project is time-consuming. Hence, the temporal datasets' size considerably increases compared to the snapshot file, which contains only the final state report (see Table 8). This also increases the time to train the models, as we train different machine learning models in different configurations (to be explored in Subsection 2.5). We had to compromise the number of projects to fit the computational power we had available. On the other hand, this number of projects allows us to do a fine-grained analysis of the results as we did in the following sections.

We also perform three filters on the `snapshot.csv` file (the file that contains the final state reports, used to create the Temporal Dataset) to remove reports that contain at least one of the following characteristics: **1)** no related commit; **2)** *RRT* = 0; **3)** reports with two or fewer states. We argue that these reports do not represent the traditional bug workflow, that would be: bug discovery, report the bug, the bug-fixing process being discussed and documented in the report (with updates on the report), the report is closed/resolved and associated with

a commit that contains the code to fix the bug. An in-depth analysis would be essential to characterize these reports with some anomalous behavior. However, we have a few hypotheses. For instance, a report created and resolved/closed instantly ($RRT = 0$) was probably registered only for documentation purposes. The reports without commit could be reported by accident. Another hypothesis would be that one notices that the reported case is not a bug, a duplicated or resolved one during the report triage. To minimize the chances of using reports that may fall into one of these cases, we chose to use only the ones that present strong evidence that has passed by a bug's natural workflow.

To check the viability of these filters, we randomly selected 30 reports (in the subset of removed reports by the filters) from each project for a total of 300 bug reports. We analyzed each of them in the Jira platform in its original format (raw data). We look for evidence that the reports represent one of the cases we suggest: not a bug, duplicated or reported by documentation proposes. We call these bugs 'non-traditional bug reports'. Their counterpart we call "normal bugs". Considering all the 300 reports, we gathered evidence that 240 of them falls into one of the cases: duplicated bug, not a bug, reported by documentation proposes (already fixed), already fixed by previous versions, imported from another source (the discussion and original report was made in GitHub or mail list, not in Jira), stale bugs (reported a long time ago and already fixed in posterior versions), reported with a solution (the patch that solves the problem was uploaded in minutes after the report creation, between 2 to 5 minutes, indicating that the reporter founded the bug, creates the report and already upload a patch, that eventually was accepted), typos and documentation bugs (that do not demands a commit). The great majority of the normal bugs fall in the cases with reports with no commit. In some cases like Hadoop Core, they are old bug reports (from 2008/2009), or there was a comment indicating that the fixing commit was not associated. Finally, we provide a complete table with the analyzed reports and a commentary about them in the replication package[1].

Table 8 presents some of the dataset characteristics. The first column has the names of the projects. The second, the number of reports on the original `snapshot` file, as proposed in the original dataset. In the third column, we have the number of reports after applying the creation process, as described in Section 3.1.1. The fourth, fifth, and sixth columns show the number of reports caught by the three filters explained above. "Selected Reports" shows the number of unique reports selected from the `snapshot.csv` file. The last column presents the

[1] https://zenodo.org/record/5338495#.YS0bdVtv9H4

total numbers of states created from the Selected Reports.

Table 8 – Projects datasets information after applying filters to remove 'non-traditional' bug reports.

Project	No. of Reports (snapshot file)	No. of Reports (temporal dataset)	RRT = 0	No Commit Associated	No. of States ≤ 2	Selected Reports	All Reports States (temporal dataset)
Flink	3317	31290	659	928	137	2188 (65.96%)	25915
Hadoop Core	2861	44717	65	705	0	2116 (73.96%)	36794
Hadoop HDFS	3214	55852	53	666	0	2525 (78.56%)	46845
Hadoop Mapreduce	2210	34021	64	866	0	1311 (59.32%)	22967
Hadoop Yarn	2090	41946	12	103	0	1983 (94.88%)	40355
Kafka	2404	21489	61	462	19	1891 (78.66%)	17952
Lucene	2004	21943	182	153	14	1671 (83.38%)	19935
Solr	2249	25101	161	255	32	1821 (80.96%)	22431
Spark	6380	49438	66	604	101	5640 (88.40%)	45127
Zookeeper	882	16823	22	107	4	755 (85.60%)	15384

The original `snapshot.csv` file contains 53 attributes, but we only use a subset of these alongside some attribute variations. Table 9 shows each of the 18 attributes we employ and their description. We selected those attributes based on two reasons: **1)** they are easy to compute; **2)** they are effortless attributes, which is ideal for a first proposal. Since the final goal is a program that estimates the report resolution time at any moment of its life cycle, the model could benefit from easy computing and acquiral of report features. From a machine learning perspective, we usually train the initial models with simple and easy to compute attributes. Afterward, we will try more complex models, algorithms and attributes. We also transform the textual fields 'Comments', 'Description', and 'Summary', in features using Bag of Words (BoW) technique, and trained models logistic and neural networks models using three different sets of features: i) only the textual information as BoW; ii) only the ones presented in Table 9; iii) and a hybrid approach, where we use the combination of both features groups. For all projects and models, the best results were acquired using only the ones presented in Table 9 (these results can be found in the replication package). For future works, we intend to perform a more detailed analysis of the textual fields and evaluate the relevance of the textual fields with different Natural Language Processing techniques.

3.1.4 Models training methodology

We choose three machine learning methods to create models using the temporal dataset: logistic regression, MultiLayer Perceptron (MLP), and Gaussian process. For the

Table 9 – Dataset features description.

Attribute Name	Description	Possible Values	Addition Information
NoAttachedPatches	Number of patches attached to the report	ℕ	Same idea of the original
NoAttachments	Number of files attached to the report	ℕ	Same idea of the original
NoComments	Number of comments in the report	ℕ	Same idea of the original
Priority	Report priority label encoding	{1,2,3,4,5}, meaning, respectivily, Trivial, Minor, Major, Critical and Blocker	Original priority field values mapping of the original dataset values;
Status	Inform the report status in a one-hot-encoding representation.	{0,1}, the features are 'Open', 'In Progress', 'Reopened', 'Resolved', 'Patch Available' and 'Closed'.	Status field one-hot-encoding of the original dataset values;
NoAffectedVersions	Number of system versions affected by the bug	ℕ	A simplification of the original dataset "AffectsVersions" field.
HasAssignee	Inform if the report has a associated assignee	{0,1}	Binary attribute derivated from the "Assignee" field in the original dataset.
NoComponents	Number of components affected by the bug	ℕ	Binary attribute derivated from the "Components" field in the original dataset.
NoDescriptionTopWords	Number of Top 1000 most frequent words of a bug detailed description	ℕ	A simplification of the original dataset "DescriptionTopWords" field.
UniqueNoDescriptionTopWords	Number of unique Top 1000 most frequent words of a bug detailed description	ℕ	A simplification of the original dataset "DescriptionTopWords" field.
NoSummaryTopWords	Number of Top 1000 most frequent words of a brief one-line bug summary	ℕ	A simplification of the original dataset "SummaryTopWords" field.
UniqueNoSummaryTopWords	Number of unique Top 1000 most frequent words of a brief one-line bug summary	ℕ	A simplification of the original dataset "SummaryTopWords" field.
NoCommentsTopWords	Number of Top 1000 most frequent words of a bug detailed summary	ℕ	A simplification of the original dataset "CommentsTopWords" field.
UniqueNoCommentsTopWords	Number of Top 1000 most frequent words of a bug detailed summary	ℕ	A simplification of the original dataset "CommentsTopWords" field.
TotalLinks	The number of other issue reports linked to the report.	ℕ	A aggregation of the original dataset "InwardIssueLinks" & "OutwardIssueLinks" fields.
DSLU	Days Since the Last report Update	ℕ	- Created on the temporal dataset process creation. - The report's idle time between a previous and a current state
NumberOfUpdates	Number of updated fields since the last report state	ℕ	- Created on the temporal dataset process creation. - Represents the number of features with different values bettween a previous and a current state
State	Report State number	ℕ	- Created on the temporal dataset process creation.
Progress	Actual report state divided by the number of report states	{0...1}	Used in results analysis
ResolutionTimeInDays	The report resolution time in days.	ℕ	- Dependent variable that the models try to predict;

logistic regression we use the sklearn[2]. For the Gaussian process model, we use the GPFlow[3] implementation. For the MLP, we consider the Keras library[4]. All models were trained using a 5-fold data partition to perform cross-validation. More details on how we perform this partition in subsection 3.1.5

We test different choices for two model configurations, namely: output format (i.e., the way to estimate the RRT, y in machine learning terms); and how to deal with class imbalance (to use or not minority class over-sampling or majority class under-sampling). For the output format, we evaluate two ways to estimate the RRT: "two labels (threshold = 5 days)", where the reports are grouped by in two intervals: $[0,5[$, $[5,\text{inf}]$; and "two labels (threshold = 10 days)", where the reports are grouped by in two intervals: $[0,10[$, $[10,\text{inf}]$. The numbers inside the intervals are the real RRT calculated as explained in Definitions 1 and 3. The idea to test two thresholds is to verify the model's viability to help estimate in short or medium/long-term releases. Seven and fourteen days seem to be a natural choice, as they are the most common period sizes of sprints. We tested several thresholds (5, 7, 10, 14, and 15 days) in a previous round of experiments using logistic regression (the complete results table can be found in the replication package). The results indicate that smaller thresholds present better results than more significant thresholds. We choose five and ten days to present the best results overall without losing the idea to verify the model's viability to help estimate in short or medium/long-term releases. Also, five and ten days can be seen as one or two weeks in terms of business/working days. We show the label distribution for each project in Table 10.

We summarize our models as follows. We combine the three aforementioned models, two ways to model the RRT, and four approaches to deal with the class imbalance (two under-sampling methods, over-sampling or none), which results in 24 ($3 \times 2 \times 4 = 24$) classification models for each project. We use the following rule to refer to each model — [model] [y_format] [balance_data] — where:

- model: 'logreg' for logistic regression, 'gp' for gaussian process and 'nn' for neural networks (MLP).
- y_format: 'two_labels_5' for the two intervals RRT estimation with threshold=5 and 'two_labels_10' for the two intervals RRT estimation with threshold=10.
- data: The use or not of class balancing strategies. Original Data (OD) means using the

[2] https://scikit-learn.org/
[3] https://www.gpflow.org/
[4] https://keras.io/

Table 10 – Labels distribution. The table shows the number of reports, by projects, of each specific RRT interval.

	Threshold = 5 days		Threshold = 10 days	
	[0, 5[	[5, inf]	[0, 10[	[10, inf]
Flink	14769	11146	17814	8101
Hadoop Core	19867	16927	24137	12657
Hadoop HDFS	26173	20672	32005	14840
Lucene	14103	5832	15522	4413
Hadoop Mapreduce	12306	10661	14994	7973
Spark	26753	18374	32076	13051
Hadoop Yarn	21749	18606	27001	13354
Zookeeper	5643	9741	7134	8250
Kafka	9116	8836	11174	6778
Solr	11688	10743	13531	8900

original data, Synthetic Minority Over-sampling Technique (SMOTE) (CHAWLA *et al.*, 2002) implies the use of oversampling of minority classes. Cluster Centroids (CC) implies the use of Cluster Centroids to under-sampling the majority classes, while RND implies the use of random under-sampling of majority classes data points.

For instance, a model named 'logreg_two_labels_5_SMOTE' indicates a logistic regression model with SMOTE used to over-sample the minority class, and used to predict if a given report will take more or less than 5 days to be resolved/closed.

The machine learning methods parameters used to train the models are as follows. For the logistic regression, we use the library default values. For the MLP, we tested a few architectures and noticed that two hidden layers with 128 neurons each performed better. For the gaussian process models, due to the dataset sizes, we use a stochastic variational inference procedure (HENSMAN *et al.*, 2013).

3.1.5 The Train/Test Split Method

In this subsection, we explain in detail how we create the folds to train and test the models. Initially, we have two rounds of experiments. The first one is the baseline approach, where we train the data using the work of Zhang *et al.* (2013), in three different reports' states scenarios (EXP 1, 2, and 3). The second one is our approach, where we use all states in several machine learning configurations (models, class split days threshold, and data balance strategies).

Before any round of experiments, we split each data project in 5-folds. However, we

must respect two constraints when creating the train/test folds partitions:

- The different states (data points) of a report must be at the same fold partition (i.e., given a report, all its states must be at train fold partition OR test fold partition, exclusively).
- Different reports have a different number of states, some with more updates and others with fewer updates. We must monitor how reports with several states impact the models' performance and avoid that groups of reports with several updates end up in the same train or test split. This may cause the over-representation of a report to surpass the influence in the model of reports with a fewer number of updates.

We use the following strategy to respect both restrictions:

- For each project, we first sort all bug reports by their number of states in non-decreasing order. It gives us a list-like data structure with the reports with a higher number of updates at the beginning of the structure and the ones with fewer updates at the end.
- We split the sorted list of reports into buckets of reports, each bucket containing five reports maximum, following the order of reports presented in the list-like data structure.
- Each bucket index a report by a number $k \in \{1,2,3,4,5\}$. The index of the report indicates the partition/fold where the report goes into.

Following this strategy, we attend to both restrictions and maintain the size of each partition comparable. For 'Restriction 1', when using 5-fold (Cross-Validation (CV)) to train the models, a model k is trained with all folds but k, and it is tested with the fold k. Hence, different states of the same report are never into distinct train/test folds; for 'Restriction 2', once we split the ordered report into buckets, each bucket contains a group of reports with a similar number of updates. Hence, each fold contains an approximated representation of different categories of reports (reports with several updates and the ones with fewer updates). The size of each CV partition can be found in the replication package, where we show that each fold has a comparable size. We summarize the process in Fig. 11, which contains a real values example of the process applied to the project Spark.

3.2 Results

3.2.1 Field Changes Analysis and (ZHANG et al., 2013) replication (baseline)

Table 11 shows the most common field changes in bug reports of the ten projects we investigate. Each column contains the information for a specific project, while each line

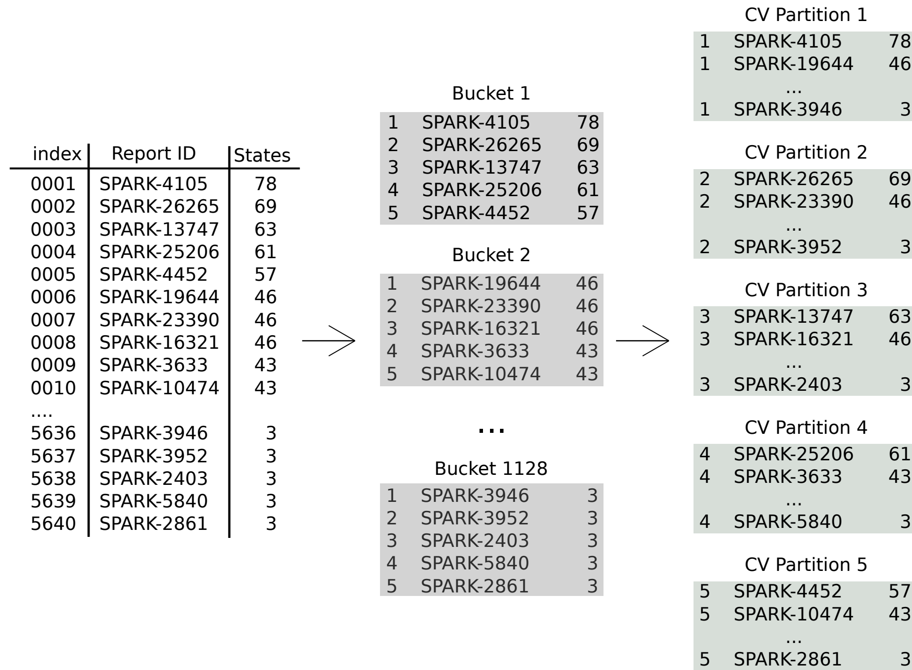


Figure 11 – The train/test 5-fold split method. The Figure presents an applied example using data from the Spark project.

represents a feature in the bug reports. The values in the Table show percentile representation of all project's bug reports with at least one field update. The Table is presented as a simplified heatmap, where the values relate to gray's intensity in each cell, with four groups of values: 0-25%, 25-50%, 50-75%, and 75%-100%. Next to each field name, we indicate the related work that uses the field (or some attribute derived from it) according to the following symbols: ★ represents our approach; Zhang *et al.* (2013) ⧫; Assar *et al.* (2016) •; Al-Zubaidi *et al.* (2017) ■. We do not indicate the work by Habayeb *et al.* (2018) because their approach does not use the fields' values but their changes, as it uses a Hidden Markov Model.

Table 11 – Top bug reports' fields updates of 10 selected projects. The table shows a simplified heatmap highlighting the most common field updates.

	Flink	Hadoop Core	Hadoop HDFS	Hadoop Mapreduce	Hadoop Yarn	Kafka	Lucene	Solr	Spark	Zookeeper
Assignee ★ ⧫	42.24%	51.28%	45.02%	53.08%	51.24%	50.00%	43.16%	62.21%	87.54%	67.01%
Attachment ★	14.32%	95.77%	97.57%	93.94%	98.80%	34.23%	83.58%	74.70%	4.70%	81.07%
Component ★	26.08%	22.96%	17.36%	16.56%	9.42%	5.40%	6.23%	9.69%	10.55%	9.75%
Description ★ ■	14.50%	17.13%	19.66%	12.49%	23.35%	19.97%	9.53%	15.56%	28.01%	11.56%
Link ★	11.28%	53.58%	52.30%	40.63%	45.12%	21.38%	17.96%	39.08%	25.03%	31.41%
Priority ★ ⧫	9.19%	11.15%	11.29%	14.03%	15.79%	12.15%	4.79%	7.24%	15.11%	14.40%
Summary ★ ⧫ •	7.23%	21.71%	26.60%	14.30%	34.88%	10.73%	8.08%	16.10%	15.66%	9.41%
Version ★	10.97%	30.55%	30.49%	30.72%	19.14%	10.32%	11.28%	14.05%	11.10%	20.52%

The most common fields updates depend on the project. We notice that the fields `Assignee`, `Attachment`, and `Link`, and are commonly updated. Other fields, such as `Summary`, `Priority` and `Description`, are used in two or more approaches, but they have a lower number

of updates when compared to the previously mentioned fields. We consider several attributes in our approach that are present in the most common field updates. This justifies our interest in working with fields values of different moments of the reports' life cycle.

Given that field updates occur in several bug reports, how do those changes impact the time fix estimation models' reliability? We address this question in the experiments described in Subsection 3.1.2. Table 12 shows the obtained results. As follows, we recap the experimental scenarios. 1) First experiment: we train and test using only the final state reports (`EXP1`); 2) Second experiment: we train and test using only the initial state reports (`EXP2`); 3) Third experiment: we train using final state reports and test with initial state reports (`EXP3`). The first column in Table 12 indicates the threshold used to split the data into two classes. The number in parentheses represents the `time unit` in days. The other three columns show the Accuracy (ACC) and F1-Score (F1) for the three data experiments. The best results are in **boldface**. It is important to note that the classes are not balanced for all the models, except when the `time unit` threshold is equal to one. Thus, the F1 values are of main concern. We tested several unit values (as presented in the baseline paper), but we only show three of them due to space constraints. They are the unit values corresponding to the 5 and 10 days (the thresholds we use in our approach) and the 1 unit value, representing each project's RRT median value. For those interested in checking all the values, we refer the reader to the replication package, where we present the values for all thresholds.

A few points can be verified after analyzing the results. First, as the time unit increases, the models' performance decrease. Second, the results present little variation for the different scenarios. Such behavior probably occurs because the attributes used in this approach present a low field change rate. `Priority` varies between 4% and 16% and `reporter` does not appear in the most common changes. `Assignee` (between 42% and 87%) and `Summary` (between 7% and 34%) present higher change rates, but the low results variation may indicate that they are not very relevant for the models. It is intriguing that for some projects, such as Hadoop Core, Hadoop HDFS, and Spark, the best results are obtained in the EXP3 scenario. One hypothesis is that the low rate of the attributes' updates may not significantly impact the models' results. Hence, to verify how the approach performs in a scenario with attributes containing more historical updates, Table 13 shows the results using *Set 2* (see Table 7). The experiment uses the same data unit splits and data scenarios but with different attributes. The best results are highlighted in **boldface**.

Table 12 – Baseline results in different data scenarios: attributes Set 1.

Threshold	EXP1		EXP2		EXP3	
	ACC	F1	ACC	F1	ACC	F1
Flink						
Unit_0.698 (5 days)	0.5658	0.6265	0.5740	**0.6446**	0.5603	0.6284
Unit_1 (7.17 days)	0.5791	0.5492	0.5645	**0.5529**	0.5476	0.5320
Unit_1.4 (10 days)	0.5736	0.4618	0.5759	**0.4750**	0.5521	0.4288
Hadoop Core						
Unit_0.723 (5 days)	0.5387	0.5666	0.5539	0.5871	0.5586	**0.6030**
Unit_1 (6.92 days)	0.5312	0.4914	0.5355	0.5100	0.5487	**0.5364**
Unit_1.45 (10 days)	0.5614	0.4179	0.5591	0.4391	0.5605	**0.4613**
Hadoop HDFS						
Unit_0.759 (5 days)	0.5901	0.6097	0.5968	0.6267	0.5949	**0.6319**
Unit_1 (6.58 days)	0.5945	0.5603	0.5881	0.5707	0.5909	**0.5834**
Unit_1.52 (10 days)	0.6067	0.4770	0.5976	**0.4960**	0.5885	0.4929
Kafka						
Unit_0.638 (5 days)	0.6060	0.6580	0.6034	**0.6629**	0.5912	0.6586
Unit_1 (7.83 days)	0.5875	0.5595	0.6008	**0.5916**	0.5711	0.5647
Unit_1.28 (10 days)	0.5970	0.5189	0.6060	**0.5556**	0.5843	0.5283
Lucene						
Unit_1 (1.79 days)	0.5464	**0.5342**	0.5368	0.5170	0.5368	0.5161
Unit_2.79 (5 days)	0.6242	**0.3866**	0.6074	0.3467	0.6206	0.3402
Unit_5.58 (10 days)	0.6708	0.2663	0.6786	**0.2714**	0.6882	0.2657
Mapreduce						
Unit_0.55 (5 days)	0.6003	0.6663	0.6133	**0.6869**	0.5828	0.6627
Unit_1 (9.08 days)	0.5858	0.5525	0.5797	**0.5678**	0.5485	0.5173
Unit_1.1 (10 days)	0.5759	0.5238	0.5683	**0.5384**	0.5332	0.4804
Solr						
Unit_0.591 (5 days)	0.5524	0.5982	0.5524	**0.6037**	0.5371	0.5926
Unit_1 (8.46 days)	0.5458	0.5199	0.5420	**0.5255**	0.5310	0.5121
Unit_1.18 (10 days)	0.5491	**0.5012**	0.5338	0.4923	0.5343	0.4864
Spark						
Unit_1 (4.4 days)	0.5887	0.5573	0.5661	0.5458	0.5287	**0.5668**
Unit_1.14 (5 days)	0.5881	0.5320	0.5674	0.5184	0.5273	**0.5467**
Unit_2.27 (10 days)	0.6415	**0.3918**	0.6248	0.3771	0.5445	0.3909
Yarn						
Unit_0.612 (5 days)	0.5951	0.6536	0.6001	**0.6665**	0.5789	0.6518
Unit_1 (8.17 days)	0.5754	0.5491	0.5724	0.5511	0.5573	**0.5527**
Unit_1.22 (10 days)	0.5890	0.5254	0.5855	**0.5265**	0.5633	0.5250
Zookeeper						
Unit_0.227 (5 days)	0.7152	0.8165	0.7192	0.8188	0.7272	**0.8283**
Unit_0.455 (10 days)	0.6371	0.7212	0.6424	**0.7289**	0.6265	0.7264
Unit_1 (22 days)	0.5589	0.5340	0.5404	**0.5361**	0.5258	0.5147

Table 13 – Baseline results in different data scenarios: attributes Set 2.

Threshold	EXP1		EXP2		EXP3	
	ACC	F1	ACC	F1	ACC	F1
Flink						
Unit_0.698 (5 days)	0,5813	0,6431	0,5731	**0,6455**	0,5425	0,5970
Unit_1 (7.17 days)	0,5777	0,5622	0,5731	**0,5769**	0,5553	0,5276
Unit_1.4 (10 days)	0,5823	0,5067	0,5795	**0,5259**	0,5603	0,4642
Hadoop Core						
Unit_0.723 (5 days)	0,6077	**0,6395**	0,5728	0,6176	0,4976	0,4414
Unit_1 (6.92 days)	0,6011	**0,5854**	0,5685	0,5669	0,5208	0,4021
Unit_1.45 (10 days)	0,6040	**0,5212**	0,5756	0,4972	0,5496	0,3613
Hadoop HDFS						
Unit_0.759 (5 days)	0,6701	**0,6979**	0,6059	0,6470	0,5438	0,5071
Unit_1 (6.58 days)	0,6653	**0,6587**	0,6016	0,6023	0,5117	0,2599
Unit_1.52 (10 days)	0,6562	**0,5807**	0,6048	0,5287	0,5671	0,2218
Kafka						
Unit_0.638 (5 days)	0,6076	0,6555	0,6156	**0,6776**	0,5907	0,6540
Unit_1 (7.83 days)	0,6013	0,5790	0,6023	**0,6064**	0,5595	0,5606
Unit_1.28 (10 days)	0,6129	0,5504	0,6076	**0,5792**	0,5759	0,5251
Lucene						
Unit_1 (1.79 days)	0,5907	**0,5760**	0,5326	0,5321	0,5153	0,4981
Unit_2.79 (5 days)	0,6362	**0,4428**	0,5901	0,3868	0,5948	0,3820
Unit_5.58 (10 days)	0,6834	**0,3499**	0,6427	0,2875	0,6409	0,3107
Mapreduce						
Unit_0.55 (5 days)	0,6446	**0,7066**	0,6011	0,6810	0,4722	0,4370
Unit_1 (9.08 days)	0,6293	**0,6095**	0,5607	0,5587	0,5210	0,3295
Unit_1.1 (10 days)	0,6232	**0,5873**	0,5652	0,5506	0,5340	0,3283
Solr						
Unit_0.591 (5 days)	0,6200	**0,6710**	0,5513	0,6084	0,5019	0,4953
Unit_1 (8.46 days)	0,5931	**0,5935**	0,5316	0,5273	0,5129	0,4340
Unit_1.18 (10 days)	0,5914	**0,5748**	0,5272	0,5045	0,5239	0,4285
Spark						
Unit_1 (4.4 days)	0,5832	**0,5607**	0,5640	0,5588	0,5220	0,5524
Unit_1.14 (5 days)	0,5894	**0,5431**	0,5569	0,5275	0,5184	0,5320
Unit_2.27 (10 days)	0,6383	**0,4160**	0,5950	0,3837	0,5477	0,3818
Yarn						
Unit_0.612 (5 days)	0,6289	**0,6921**	0,5683	0,6431	0,4962	0,5036
Unit_1 (8.17 days)	0,6132	**0,6052**	0,5527	0,5426	0,5119	0,4174
Unit_1.22 (10 days)	0,6092	**0,5740**	0,5507	0,5047	0,5179	0,3808
Zookeeper						
Unit_0.227 (5 days)	0,7046	0,8033	0,7073	**0,8085**	0,6305	0,7426
Unit_0.455 (10 days)	0,6649	**0,7409**	0,6371	0,7267	0,5722	0,6573
Unit_1 (22 days)	0,5854	**0,5808**	0,5483	0,5507	0,5325	0,5039

With more attributes, the results are different. It is noticeable that the new attributes improve the models' results for all the cases. In all projects, there is a significant performance drop in the EXP3 scenario. This shows that the initial and final reports are different enough to drop the model's performance, which indicates that bug reports' updates impact the model performance in all projects. In all unit values, EXP1 and EXP2 consistently present considerable higher values compared to the EXP3.

We can now address our first research question **RQ1: How frequent are the bug reports' fields updates, and how these updates impact fixing estimation models?** *Answer: We verify that bug reports fields' updates are common across the ten different projects and impact fixing estimation models in all projects we test.* The use of inappropriate report states (*i.e.* last states reports) to train the models can provide more optimistic results (between 0.01 to 0.4 in F1 absolute values, depending on the project and threshold values) than using the initial report states.

3.2.2 Training models with all bug reports states

Table 14 shows the five-folds average results for the best models configuration of each machine learning algorithm applied individually for each project, using the temporal dataset, respectively: Flink, Hadoop Core, Hadoop HDFS, Lucene, Hadoop Mapreduce, Spark, Hadoop Yarn, Zookeeper, Kafka, and Solr. We evaluate the models using five metrics: Log-Loss (LGL), accuracy (ACC), f1-measure score (F1), Precision (PRC), and Recall (RCL). We highlight the best results in **boldface** for each project and use them to perform the analysis and answer the RQs.

We acquire the best results by classifying the reports into two classes, with five days threshold (Logistic regression, Neural Network, and Gaussian Process present similar metrics' values for the majority of projects). As one can see, all the best models, expect Zookeeper, use some data balancing strategy. For six projects, the cluster-centroids under-sampling technique presents the best results, while for two other projects, the random under-sampling presents the best values. For Hadoop HDFS, Hadoop Mapreduce, Kafka, and Spark projects, Gaussian Process provides the best results, using an under-sampling approach. For Hadoop Mapreduce and Hadoop Core, Flink, Lucene and Solr projects, the Logistic Regression using some under-sampling approach presents the best results. The neural networks present best results for the projects Yarn and Zookeeper, using an over-sampling technique and the original data, respectively.

Table 14 – All projects overall best results.

Model	ACC	F1	PRC	RCL	LGL	ROC
flink_logreg_two_labels_5_CC	0.5737	**0.5800**	0.5026	0.6863	0.6608	0.6476
flink_gp_two_labels_5_SMOTE	0.6139	0.5530	0.5519	0.5621	0.6513	0.6627
flink_nn_two_labels_5_SMOTE	0.5852	0.5536	0.5164	0.6028	0.6587	0.6179
hadoop_logreg_two_labels_5_CC	0.6054	**0.5756**	0.5691	0.5828	0.6648	0.6412
hadoop_gp_two_labels_5_CC	0.5748	0.5727	0.5394	0.6534	0.6793	0.6326
hadoop_nn_two_labels_5_OD	0.6053	0.5157	0.5891	0.4643	0.6617	0.6362
hdfs_logreg_two_labels_5_RND	0.6124	0.5661	0.5592	0.5733	0.6532	0.6542
hdfs_gp_two_labels_5_CC	0.5725	**0.5945**	0.5134	0.7161	0.6968	0.6446
hdfs_nn_two_labels_5_SMOTE	0.5857	0.5425	0.5280	0.5590	0.6524	0.6106
lucene_logreg_two_labels_5_RND	0.6114	**0.4720**	0.3921	0.5956	0.6581	0.6417
lucene_gp_two_labels_5_RND	0.3786	0.4562	0.3131	0.8897	0.7545	0.6041
lucene_nn_two_labels_5_CC	0.5123	0.3620	0.2933	0.4741	0.9287	0.5001
mapreduce_logreg_two_labels_5_CC	0.6100	0.6053	0.5702	0.6455	0.6529	0.6626
mapreduce_gp_two_labels_5_CC	0.5569	**0.6255**	0.5295	0.8259	0.6894	0.6564
mapreduce_nn_two_labels_5_SMOTE	0.5945	0.5876	0.5566	0.6258	0.8741	0.6407
spark_logreg_two_labels_5_RND	0.6416	0.6228	0.5450	0.7269	0.6159	0.7107
spark_gp_two_labels_5_CC	0.5776	**0.6284**	0.4897	0.8784	0.6708	0.6905
spark_nn_two_labels_5_SMOTE	0.6530	0.6075	0.5635	0.6600	0.6412	0.7115
yarn_logreg_two_labels_5_CC	0.5929	0.5707	0.5554	0.5881	0.6626	0.6398
yarn_gp_two_labels_5_CC	0.6069	0.5677	0.5768	0.5762	0.6604	0.6540
yarn_nn_two_labels_5_SMOTE	0.5861	**0.5832**	0.5456	0.6321	0.6426	0.6257
zookeeper_logreg_two_labels_5_OD	0.6557	0.7723	0.6639	0.9266	0.6306	0.6356
zookeeper_gp_two_labels_5_OD	0.6317	0.7731	0.6339	0.9920	0.7034	0.6148
zookeeper_nn_two_labels_5_OD	0.6589	**0.7764**	0.6626	0.9404	0.6258	0.6406
kafka_logreg_two_labels_5_CC	0.6426	0.6643	0.6175	0.7192	0.6144	0.7075
kafka_gp_two_labels_5_RND	0.6337	**0.6738**	0.6052	0.7756	0.6288	0.7057
kafka_nn_two_labels_5_SMOTE	0.6351	0.6552	0.6105	0.7165	0.6300	0.7023
solr_logreg_two_labels_5_CC	0.6000	**0.6092**	0.5718	0.6530	0.6650	0.6407
solr_gp_two_labels_5_RND	0.5717	0.5466	0.5910	0.5920	0.6834	0.6437
solr_nn_two_labels_5_SMOTE	0.5998	0.5943	0.5758	0.6173	0.6769	0.6413

Notably, some models metrics (accuracy, precision, recall or log-loss) can perform somewhat better than the selected models. Nevertheless, we prefer to choose the models with higher F1 due to the data imbalance nature (see Table 10).

It is also noticeable that for the majority of the projects, the best results are not ideal for a real-world application scenario, i.e., the models could not be used in the Jira platform to perform reasonable estimations with the presented accuracy of around 0.55 ~ 0.65 and f-measure around 0.47 ~ 0.77. The only two projects that provide some interesting results are Kafka and Zookeeper with f-measure higher than 0.67 and recall values above 0.77.

In this scenario, we can address our second research question **RQ2: What is the most promising model configuration to build reliable models for fixing time estimation considering bug reports at different stages of evolution?** *Answer: with our set of experiments and*

data attributes, we verify a pattern where the most promising way to model the selected projects bug reports, taking into account their evolution, is the five-day threshold binary classification reports using an appropriate data balancing technique.

3.2.3 Models Performance by Group: Progress and Resolution Intervals

We use the following strategy to address RQ3. After the 5-fold training phase using the temporal dataset, we obtain five models for each configuration. Then we use each test set with its corresponding k-fold model after selecting the best model configuration, which gives us five accuracy values, one for each test set. Next, we calculate the average model accuracy for each group. We group the reports in two different ways: i) by their percentile of overall resolution time progress, and ii) by six different intervals, namely, [0,5[, [5,10[, [10,15[, [15,20[, [20,25[, and [25, inf] days. To calculate the report progress, we divide its current state (equal to the number of reports/changes until its current state plus one) by the report states' total number, which gives a value between 0 and 1. Since we have the true resolution interval of each report, the interval calculation group can be obtained directly. We expect that such progress information will enable an overall view. The intervals provide us another level of granularity since we classify reports with RRT ranges from a few days up to more than a hundred days. For instance, there are reports with RRT values greater than 100 and others with lesser than 5. Thus, the reports' performance over specific RRT intervals may indicate tendencies difficult to notice only from the progress information. Fig. 12 summarizes the whole process. Figures 13 and 14 shows the accuracy values for each project for the two reports groups. We use the models in **boldface** (i.e. the best models in terms of F1-measure values) in the result's Table 14, to perform the analysis.

The Fig. 13 shows, by project, how good the accuracy of the selected models (highlighted in the results tables) is to estimate the report resolution time by its evolution progress. In other words, the progress tells us how close the classified report state is to its initial state (when it was opened, with smaller evolution progress values) or to its final state (when it was closed, higher evolution progress values). When we analyze the Fig. 13 we notice a particular behavior: the estimations start with values around 0.7 and 0.9 in progress 0.1, they drop with the report progress increases and raise close to the progress 1. The only exception is Lucene, which starts with an accuracy value above 0.6 and increases until reaching values close to 0.8 at progress 1. This behavior varies in intensity depending on the project. The better performance on reports close to resolution is due probably one reason: the report's status

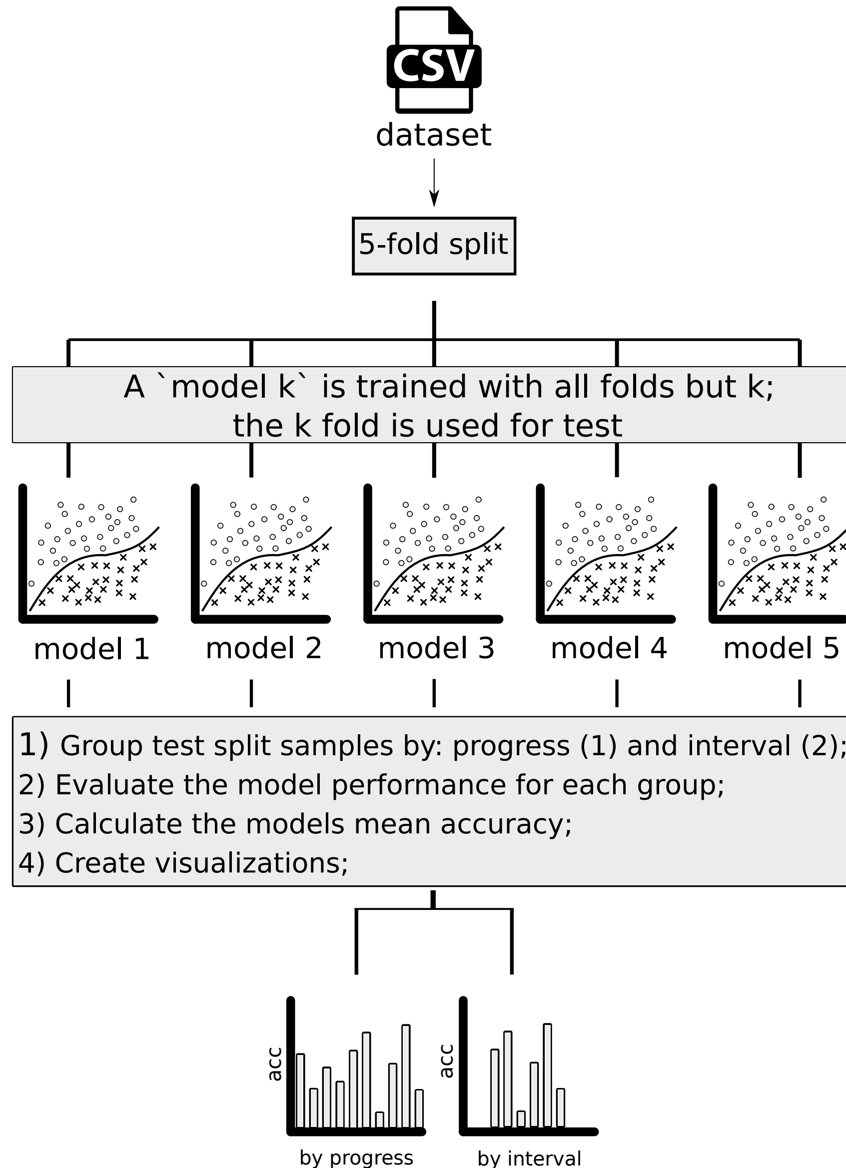


Figure 12 – Workflow to report RRT evaluation by progress and interval.

attribute to be classified. The status attributes are very descriptive attributes since its one-hot encoding representation contains the values “closed” and “resolved”, which means these reports mostly will have $RRT = 0$. We say mostly because sometimes a report is re-opened, which may indicate that the report was wrongly closed. Furthermore, sometimes the project manager must perform a confirmation step, changing the report status from resolved to closed, delaying the final report resolution. Thus, these are two attributes that are highly correlated with the report resolution, but they depend on each report's evolutional context. Once we are dealing with each report independently, the models do not have this evolutional context, making the final report's accuracy not perfect. The good accuracy in these close-to-the-final states reports is not that interesting in real-world scenarios. On the other hand, we see that 0.1 progress values present slightly higher values than the posterior ones (except 1.0 values, explained earlier). We propose

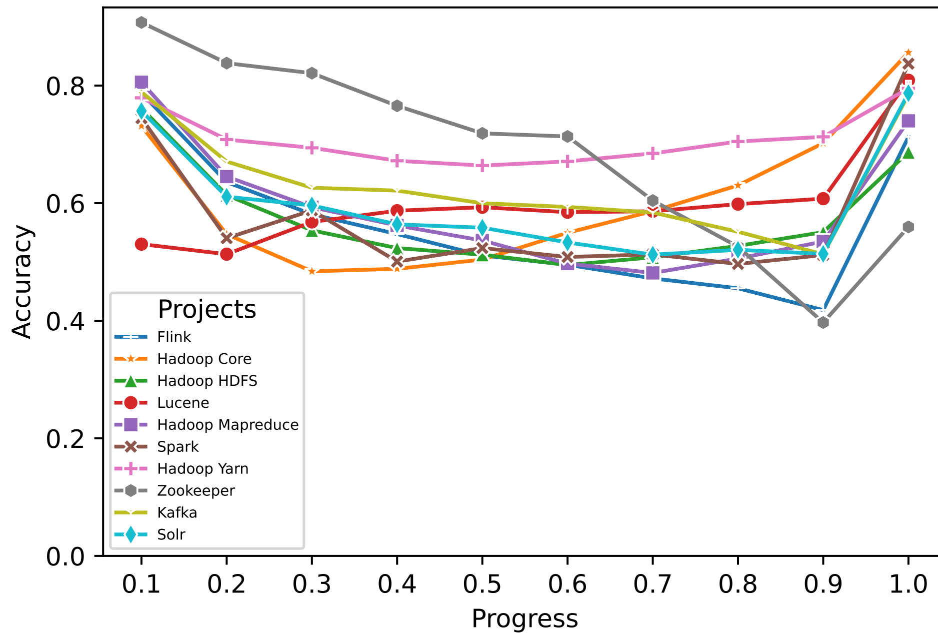


Figure 13 – Accuracy evaluation by report resolution progress. The intermediate reports' accuracy is lower than the accuracy of initials and finals states reports.

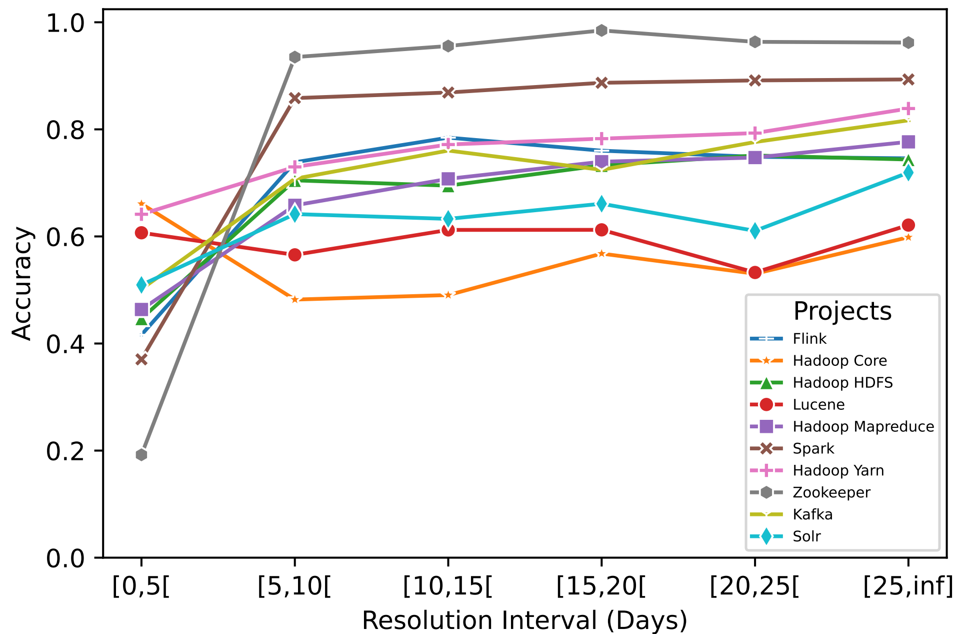


Figure 14 – Accuracy evaluation by report resolution interval. The most difficult reports to classify in most projects are those with RRT in the interval between zero and five days.

to build models that predict the RRT at any moment of the report life cycle. However, if we can provide reasonable estimations at the initial or a set of initial reports states (for instance, up to n initial states), the further estimations for future reports states becomes unnecessary since the

following estimations could be calculated based on the initial one.

We cannot say that all initial state reports are present at the 0.1 progress group because it depends on each bug report's number of states. For instance, a report with five states may have its initial state in the 0.2 progress group. However, this tendency raises another question: how good are the models to predict the initial report state? This is an interesting subject to evaluate because of two reasons: the RRT value of an initial state is the *real* report RRT (see Definition 1), once the posterior states RRT are based on the initial one; and if one can predict with a good rate the initial reports, we can use this estimation, and the estimations of the posterior states become unnecessary. To answer this question, we perform another analysis and verify the accuracy for each model/project in the first five states reports. Table 15 shows the model's performance classifying all reports (same as the selected models at Tables 14) and classifying the first up to fifth report states. The states values are cumulative, so 'state one' results represent the evaluation only of the initial report state, 'up to 2' results for the initial and the second report state, 'up to 3', the initial, second, and third state, and so on so forth.

The values in Table 15 show that the models present higher f-measure and recall values predicting the initials reports than when predicting all reports. We see a decrease in the accuracy values and precision in some projects, but, again, due to data imbalance, we prefer to look at the f-measure.

Those are promising results. We have a set of models using simple attributes to predict RRT of initial states with an F1 around 0.63 up to 0.87. The only exception is the project Lucene, which does not provide a significant improvement in the first three states. In this scenario, we do not have problems with the report's evolution (i.e., fields changes and addition) and a set of attributes easy to compute. It is also essential to notice that our approach presents better results when estimating the fixing resolution time using the initial states when compared with the best results presented in the baseline approach (EXP2 and EXP3, Table 13). These results also create room for a few insights that we will raise in the Discussion section. Nevertheless, for now, we can answer our third research question **RQ3: To what extent is there a moment in the bug report life cycle where a resolution estimation is more precise?** *Answer: with our set of experiments and data attributes, we verify a pattern where the most promising way to predict the selected projects bug reports bug-fixing time is at their initial states, with better results than when we try to predict all states.*

To conclude the analysis, the Fig. 14, shows the models' accuracy in a few RRT

Table 15 – Models results classifying initial states reports.

Project	Up to state	ACC	F1	Precision	Recall
Flink	1	0.5941	0.7126	0.5857	0.9099
	2	0.5961	0.7062	0.5873	0.8861
	3	0.5722	0.6975	0.5564	0.9345
	4	0.5645	0.6834	0.5572	0.8839
	5	0.5807	0.7017	0.5707	0.9108
	All States	0.5737	0.5800	0.5026	0.6863
Hadoop Core	1	0.5499	0.6815	0.5441	0.9158
	2	0.5594	0.6880	0.5552	0.9096
	3	0.5951	0.7244	0.5979	0.9222
	4	0.5868	0.7115	0.5856	0.9113
	5	0.5304	0.6575	0.5220	0.8929
	All States	0.6054	0.5756	0.5691	0.5828
Hadoop HDFS	1	0.5640	0.6798	0.5531	0.8864
	2	0.5667	0.6882	0.5561	0.9088
	3	0.5655	0.6797	0.5610	0.8682
	4	0.5586	0.6856	0.5529	0.9069
	5	0.5767	0.6890	0.5723	0.8733
	All States	0.5725	0.5945	0.5134	0.7161
Lucene	1	0.5323	0.4469	0.3474	0.6291
	2	0.5124	0.4540	0.3428	0.6782
	3	0.5555	0.4430	0.3765	0.5420
	4	0.5821	0.4740	0.4085	0.5706
	5	0.4946	0.4955	0.3698	0.7534
	All States	0.6114	0.4720	0.3921	0.5956
Hadoop Mapreduce	1	0.6166	0.7479	0.6176	0.9487
	2	0.5689	0.7087	0.5622	0.9584
	3	0.6023	0.7417	0.6003	0.9704
	4	0.6269	0.7632	0.6294	0.9693
	5	0.6057	0.7450	0.6008	0.9805
	All States	0.5569	0.6255	0.5295	0.8259
Spark	1	0.5199	0.6339	0.4757	0.9500
	2	0.5141	0.6307	0.4775	0.9302
	3	0.5164	0.6356	0.4849	0.9222
	4	0.5072	0.6327	0.4719	0.9596
	5	0.4922	0.6229	0.4623	0.9547
	All States	0.5776	0.6284	0.4897	0.8784
Hadoop Yarn	1	0.6009	0.7159	0.6349	0.8206
	2	0.6873	0.7782	0.6729	0.9227
	3	0.6677	0.7577	0.6548	0.8990
	4	0.6673	0.7592	0.6555	0.9021
	5	0.6262	0.7261	0.6218	0.8723
	All States	0.5861	0.5832	0.5456	0.6321
Zookeeper	1	0.7479	0.8545	0.7488	0.9951
	2	0.7338	0.8442	0.7342	0.9933
	3	0.7048	0.8251	0.7049	0.9952
	4	0.7853	0.8780	0.7834	0.9989
	5	0.7010	0.8220	0.7024	0.9914
	All States	0.6589	0.7764	0.6626	0.9404
Kafka	1	0.5904	0.7028	0.5588	0.9471
	2	0.6195	0.6274	0.6524	0.6046
	3	0.6544	0.7101	0.6491	0.7846
	4	0.6371	0.7227	0.6325	0.8431
	5	0.6380	0.7330	0.6101	0.9181
	All States	0.6337	0.6738	0.6052	0.7756
Solr	1	0.5907	0.6931	0.5771	0.8700
	2	0.5795	0.6628	0.5702	0.7954
	3	0.5891	0.6982	0.5704	0.9003
	4	0.6010	0.7071	0.5914	0.8815
	5	0.5916	0.6719	0.5783	0.8078
	All States	0.6000	0.6092	0.5718	0.6530

intervals. For the majority of the projects, the worst results occur in reports with RRT between 0 and 5 days, with higher values in the others intervals. The only exceptions are the projects 'Lucene' and 'Hadoop Core': both start with accuracy values around 0.6 at first internal, with some variations until they reach the same 0.6 value at the last interval. The 'Lucene' presents a smother variation, with 'Hadoop Core' being more erratic as the interval values increase. Besides that, we do not see any other trend.

3.3 Discussion

The main question we want to explore with our results is the impact of bug report fields changes and updates on reliably building bug fixing estimation models. We introduce the idea of bug report evolution and changes as bug report states. First, we verify how often the bug reports fields are updated and partially replicate a previous approach (ZHANG *et al.*, 2013) to check how it performs with the bug reports evolution and serve as a comparative baseline to our approach. We verify that the bug reports fields updates impact the models' reliability in different levels, in all projects. In our approach, we considered every state as a unique and independent report to train the models. After selecting the most promising machine learning models, we can verify their performance based on how close the best-classified reports are from their creation or resolution date. Our results present evidence that the reports' updates have an impact on the model's performance. This is important because we verify that few studies do not take the reports changes into account when building machine learning models for this problem (more in the Related Work and Comparison sections).

We first train the models and found the best configuration for our data. The Gaussian processes and logistic regression perform better in four projects data each, while the neural network, in two projects. The binary classification with a threshold of five days presents the best results. All the best models use some data balance strategy (over-sampling or under-sampling), except in project Zookeeper. After selecting the most promising results, we can discuss the impact of reports' evolution. To the best of our knowledge, this is the first work concerning report time-fixing estimation to compare these ways of grouping the reports. Also, we were not able to find other approaches using Gaussian processes for this problem. When we look at the results, it is noticeable that the best results are not ideal for a real-world application scenario due to their low metrics values (f-measure and precision around 0.5 ~ 0.7), even though these are values approximated to the ones presented in the literature.

The best results for all the projects are the binary classifications with the five-day threshold, using data balance strategies. This is a good indicator because if we think about the software development process in terms of sprints, it usually takes small chunks of time, like one or two weeks. In these scenarios, we can see the models being used to estimate sets of bugs that will probably be fixed within a sprint. We see that the neural networks, generally, perform worst than the other two machine learning algorithms. Neural networks have a high dependency on hyper-parameters (ZHANG *et al.*, 2017; ZHANG *et al.*, 2019), and we do not perform an extensive hyper-parameters search, mostly because of the dataset sizes. This research looks at the consistency between the models rather than the best models' higher values. If all models perform similarly in terms of metrics, we can argue that we reach the dataset and attributes limit. Once we better understand the reports' evolution impact on the models, we want to train models with a hyper-parameters optimization. Through the results, we conclude that the chosen attributes may not be good enough to provide reasonable estimations.

For future work, we intend to use more attributes that carry some evolutional content of previous reports (e.g, previous values of selected fields) and some attributes with more insightful meaning of the textual fields.Techniques that benefit from the data's evolutional nature (e.g Markov chains and Long short-term memory neural nets) could also be interesting approaches to explore.

After selecting the best models, we can explore the impact of the reports' changes on the models' performance. The results indicate that the best results are acquired when classifying the initial states reports compared to intermediary states reports. Up to five reports updates, we have higher F1, and recall in nine of the ten projects compared to classifying all reports' states. This seems counter-intuitive because it is reasonable to believe that any field updates in the report should provide more information to the models. Further research is needed to establish the reason for this behavior, but we have a few hypotheses to explore. The first one is regarding the independent way we consider every report's state. The performance drop can indicate that a past evolutional context is necessary, at least using our chosen attributes. The attributes as they are in any report states seem to be not enough to provide consistent estimations. The second one is related to the reports' idle time between updates. All selected projects are open-source software, and the bug fixing process and reports can be looser when compared to commercial software. In a previous dataset analysis, we verify that the time between updates can surpass days or months in some reports. This may also occur due to low priorities reports, but we intend to verify if this

is the case in the future. Once again, without an evolutional context, this could negatively impact the models' predictions. Once the initial reports have little to none evolutional context, this could also explain why their predictions perform better. The results also open the possibility to train models only with the initial set of reports once they perform better and make more sense in the bug-time-fixing process. To conclude the reflections regarding the results, we raise a hypothesis on why the performance drop in intermediary reports. Given the best results being at the initial states, the idea that posterior reports have a smaller RRT and inferior performance may indicate that they are not too much different from their previous states. We consider every update (or a set of updates in a small window of time) as a unique state, and each one of these updates impacts equality in the bug RRT decreases. However, some updates may have more (or even a *real*) impact on the RRT decrease compared to others. For instance, a new comment probably does not have the same impact as an attachment in the bug RRT estimator. The idea is to characterize an 'impactful update' that changes the original RRT estimation, defining when new updates bring new and relevant information to the bug report. This could reduce the number of states, focusing on those different from each other, improving the quality of the data, hence the results.

In this chapter, we look at the bug-fixing time as the information to be estimated and how the bug report evolution impacts reliable estimators. However, notice that this question can be applied in others bug report features to be estimated: priority, assignee, duplicated bugs and bug localization, all of those explored using bug reports in previous works (LAZAR *et al.*, 2014; EBRAHIMI *et al.*, 2019; GUO *et al.*, 2011; SHOKRIPOUR *et al.*, 2015; TIAN *et al.*, 2015). It would be essential in future works to explore how these bug reports updates impact the other features estimators, once in this research, we gathered evidence that it has a significant impact in bug-fixing time.

3.4 Threats to Validity

In this section, we list some threats to the validity of our research method. We organize threats into four groups: conclusion validity, internal validity, construct validity, and external validity, as suggested by the work of Wohlin *et al.* (2012).

A threat to conclusion validity would be few decisions regarding the adopted methodology. When we train the models, we lose evolutional information and relation between the report's states, with each state being considered an independent report. However, this approach allows inferring the bug fixing time of any report without any previous information about its

past field values. This approach is straightforward to implement and less resource-demanding. Nonetheless, we know that this level of independence between reports' states may not represent a real-world scenario, leading us to inferior results. However, we choose this approach to see its viability due to its simplicity. As we discuss, we verify that the reports' evolution does impact the model's performance metrics. For future work, we intend to use more attributes that carry information about previous states or even use models dealing with temporal changes over states.

A threat to internal validity is that the original data acquisition and the script to create the temporal dataset are susceptible to bugs. However, we take special care to use visual tools to visualize the results and minimize bugs chances in the datasets' creation and mining scripts. Another threat is the reports years' range, where few reports dated from 2009. We cannot measure the cultural bug fixing tasks difference over the years. In other words, we do not know if an older bug report can represent or is similar to the most recent ones. If the process changed or improved over time, fixing a bug with similar reports in different years can be discrepant.

As a threat to construct validity, we consider the best models as those with higher f-measure values due to data imbalance. However, depending on the context or project, it may be interesting that some class error does not have the same impact as the other one. For instance, let us consider the binary classification with a five-day threshold (class zero for less and class one for more than five days to fix). A misclassified class zero report may not necessarily mean that one could not fix the bug in less than five days. Maybe in the specific week, there were too many bug reports or less available programmers in the specific week to fix the bug. The bug could be a simple, low priority bug, competing for resources with other more urgent and complex bugs, leading to its fix delay. Once again, the bug report evolution context can play an essential role in the error analysis and it seems as a promising avenue for future work.

The original dataset comprises projects from nine categories for external validity, and all projects are open source. The selected projects cover three categories: 'big-data', 'database' and 'web-framework'. Thus, we cannot generalize the results for commercial software and software from others categories.

3.5 Related Works and Comparison

It is hard for us to compare with other researchers' approaches due to the unique way we deal with the reports states. A few works discuss the reports changes, but not in the same way we propose here, and all of them use different sets of models, reports' attributes, and different

datasets. Nonetheless, we look at all the most relevant papers with the same objective that we find in literature and propose a discussion regarding the points we believe are comparable. We look primarily at three points on each related work: the model's f-measure since it is a metric that appears in most papers; the moment in the reports' states in their life cycle that the models and predictions are made; the set of attributes used to build the models and how complicated/hard are to acquire them.

The most similar to our work is probably the paper of Habayeb *et al.* (2018). The work uses a dataset composed of Firefox bug reports from 2006 to 2014. The authors model the problem as a binary classification problem: a long time (slow) report to fix, or a short time (fast) report to fix. They highlight the fact that their work is one of the first that deals with this question, taking into account the temporal sequence and changes of the bug reports. They compare their proposal with a RRT model (ZHANG *et al.*, 2013), test several variations of the Hidden Markov Model (HMM), different train/test set sizes, and HMM temporal sequences length variations. As observation set, they use information about the reporting, assignment, comments, priority, among others. The models are evaluated by precision, recall, f-measure and accuracy metrics and present better results in comparison to previous proposals. The authors have a similar argument related to the importance of the report's temporal changes. They perform several experiments regarding the report life cycle moments to predict the fixing time, using a Firefox dataset. The most similar experiment to our approach is when they try to classify the initial reports with the first week's updates. The models present an average f-measure of 0,671, a smaller, but comparable value than the best results present in Table 15. Their proposal considers the evolutionary aspect of reports and uses easy to compute attributes (a set of possible report fields and state changes, not their values). However, we question the threshold value used in their approach. The authors use the bug report's median bug-fixing time by year as the threshold to set it as slow or fast. Even if it is a common strategy in other papers (HOOIMEIJER; WEIMER, 2007; KIM; WHITEHEAD, 2006), we question how this separation could be viable for significant median values. For instance, for the years 2007, 2008, and 2009, the Firefox dataset's median value is 194, 230, and 203, respectively. In a context to plan and estimate software releases, smaller fixed threshold values (i.e., 5, 10, 15 days) are more appropriate. Another case is that the year median bug-fixing time is a posteriori information, is only knowable after the year's end. How to build models to predict bug reports opened in the current year, for instance? What threshold to use in these cases? We argue that using a smaller predetermined threshold value is more suitable because of the points

mentioned above.

Thung (2016) propose an automatic prediction method of bug fixing effort. In that paper, however, the effort is code churn size, the number of lines of code that is either added, deleted, or modified to fix the bug. The author uses 1,029 bug reports from Hadoop-common and strut2 projects to evaluate his approach. The authors model the problem as a classification task, labeling bug fixing efforts into "high" and "low" categories. The 40 lines code churn size is the threshold used to define in which category a bug is. The features used to train a Support Vector Machine model are the textual content that appears in the summary and description fields of bug reports. The work compares the approach to the baseline model that classifies every bug as a low effort bug (i.e., the majority label) and present positive results. The research presents a 0.612 area under the ROC curve (Area Under the ROC Curve (AUC)) using both datasets to train the model, but it uses the last and closed report states information.

Assar *et al.* (2016) use clustering techniques to group bug reports through the description field. The paper works as a conceptual replication of the work by (RAJA, 2013) and an evaluation of the proposed method prediction accuracy. Along with the work by Weiss *et al.* (2007), this is one of the few papers that relies exclusively on the textual fields to come up with a prediction model. Given a new report, they predict its Defect Resolution Time (DRT) as the mean DRT of the most similar report cluster. The textual values extracted from the description are from the closed/resolved reports. It is impossible to say how different the report's initial descriptions are from the final because we do not have access to the datasets' historical data. As presented in Vieira *et al.* (2019), we could use an estimation that updates in the description field occur 18.16% of the mined Jira bug reports. The work concludes that the approach is not suitable for practical use due to poor results. In summary, the authors show that a straightforward clustering approach based on term-frequency in bug reports descriptions is not able to predict defect resolution time with reliable accuracy. This example serves as another case where these report's fields updates are not taken into account.

Al-Zubaidi *et al.* (2017) propose a multi-objective search-based approach to estimate issue resolution time. The search is oriented by two contrasting objectives: maximizing the model accuracy and minimizing the model complexity. In this case, their approach works for any issue, not only for bugs. A genetic programming approach is followed to search for a better symbolic regression model. They compare their best model with Case-based Reasoning (WEISS *et al.*, 2007), Random Forest, and Linear Regression. Their model the problem as

a regression one and show better results than random guessing, mean and median estimation, and case-based reasoning. Their approach also outperforms other machine learning methods, like linear regression and random forest. They use a small set of report fields (type - bug, task, improvement -, priority, reporter's reputation, title and description text, and their readability through the Gunning fog readability metric). They argue that the selected ones are likely to exist from the report creation, as the reporter, issue type, and the number of words in description and title. This shows the same concern we have about the difference between fields at the initial and final report states. They use datasets of five Jira projects (8.260 issues): Hadoop Common, HDFS, Yarn and Mapreduce, and MESOS. We cannot compare results with this approach since it has a broader scope (all issues, not only bugs) and different metrics, since we propose classification models and their regression models. However, the model's Mean Absolute error (MAE) high values (from 17.8 up to 33.35) may indicate that the approach is not reliable for practical purposes, even though their approach outperforms naive baselines and state-of-the-art techniques.

Hamill e Goseva-Popstojanova (2017) investigate two points regarding the bug fix task: 1) an analysis on the effort needed to fix software faults and the factors that affect it; and 2) an analysis on the prediction of the level of fix implementation effort based on the information provided in the software change requests. The paper text uses the term "fault", but from the context we can say that it is equivalent to bugs. The paper considers 1,200 failures/bugs extracted from the change tracking system of a large NASA mission. They are used to train three classification models to estimate the effort level of the fix implementation: Naive Bayes (NB), Decision Tree, and PART, a rule induction method based on partial decision trees. They use the ZeroR learner, a classifier that always predicts the majority class for any given sample. They evaluate the models with accuracy and show that their models did significantly better than the baseline.

Zhang *et al.* (2013) propose a Markov-based (Discrete Time Markov Chain model) method to predict the number of bugs that will be fixed in the future and other methods to different estimations. The dataset used is composed of three CA Technologies projects, a commercial corporation. The features used by the model are submitter, owner, severity, priority, ESC (if the bug is reported by end-users or by the QA team), category, and summary. The paper outlines highlights of the three proposed models. The first, a Markov model-based that shows a 3.72% Mean Relative Error (MRE) when predicting the number of fixed bugs in the future. The second

one, a Monte Carlo method for predicting the total time to fix a given number of bugs with a 6.45% MRE; and a KNN-based method to classify a particular bug as a slow or quick fix with an average weighted F-measure 72.45%. This work is replicated with an open-source software project, namely Firefox, by the authors AKBARINASAJI *et al.* in their work Akbarinasaji *et al.* (2018). The paper describes the same methodology, models, and methods of the original work. The proposed Firefox Markov based model to predict the number of fixed bugs in three consecutive months obtain a 1.70% MRE; The Monte Carlo simulation, to predict the fixing time for a given number of bugs achieve a 0.2% MRE; and the KNN-base model classifies the time for fixing bugs into slow and quick with a 62.69% f-measure.

Bhattacharya e Neamtiu (2011) investigate the correlation between various dependent variables (namely, number of developers, severity, attachments, and dependencies) and the bug-fix time. They define the developer's reputation and verify its correlation with the bug-fix time as well. The work's conclusion, acquired after a univariate regression test, indicates that the bug mentioned above reports do not exhibit a high correlation with bug-fix time. They suspect that successful predictions made in prior works can be justified by the problem known as "optimistic bias" in machine learning. The authors suggest that to avoid an optimistic bias problem in future works, researchers should train models with larger datasets and choose multiple applications to verify the model generalization power.

A few papers discuss the temporal and evolutive report changes but never as the main study object. The papers that use the summary and description fields, for instance, do not take into account possible changes that these fields might have. Even the state or the exact moment when the report and its features are collected generally is not evident during the dataset description. In the present work, we highlight our concern with this inherent characteristic of the bug reports life cycle.

Another noticeable thing is the relatively small variation on machine learning methods on the majority of the papers. Most works evaluate only a small number of models. In our case, we aim to diversify our experiments with a variety of models. For instance, we include in our evaluations neural networks models and gaussian processes, choices that is rarely (if ever) seen in this kind of problem, maybe due to the usually small sized datasets.

4 BAYESIAN DATA ANALYSIS APPLIED TO BUG REPORTS DATA

This chapter describes the BDA applied to all projects of our proposed dataset in Chapter 2. The main goal of this chapter is to study the relation between selected bug reports features and the bug fixing time. We choose bayesian data analysis to drawn our conclusion.

We formalize the research questions we intend to explore in this Chapter as follows. **RQ1**: *How the existence of links in bug reports impacts the BFT?* Some bug reports are related to others (see Section 2.4). This relation can assume different forms as *blocked* or *duplicates* (for instance, bug report *A* is blocked / duplicated by another bug report *B*). In this RQ, we want to explore how these relations between bug reports impact the bug fixing time. **RQ2**: *How the priority level of a bug report impacts the BFT?* Every bug report has an associated priority value. Priority is a very explored feature in bug reports research field (ALMHANA *et al.*, 2020; TIAN *et al.*, 2015; UMER *et al.*, 2018) and in this RQ, we intend to understand its role in the BFT. **RQ3**: *How the code-churn size of fixing commits relates to the BFT?* Bug-fix commits are those bringing the changes that fix a reported bug. Thus, computing the code-churn size related to a fixing commit is possible. To answer this research question, we first group the bug reports into two categories: (i) reports with low code-churn values and (ii) reports with high code-churn values (the threshold is the project's code-churn median value). Next, we evaluate if the average BFT of both groups is significantly different.

We now present the Chapter remainder. Section 4.1 presents an introductory theoretical foundation of Bayesian statistics. In Section 4.2, we discuss the selected features and how we create data groups to fit the proposed models. Section 4.3 presents the adopted process to construct and describe the models. Section 4.4 presents the results after computing the posterior distributions of the proposed models. In Section 4.5, we explore different hierarchical models compositions and discuss the results. We conclude this Chapter with Section 4.6, where we discuss the results and point to future directions, list the Threats to the Validity in Section 4.7 and the Related Works in Chapter 4.8.

4.1 Bayesian Data Analysis

In this section, we present an introduction to the BDA concepts. While we cover most of the BDA ideas applied in this chapter, we suggest additional sources for a more detailed and complete explanation of the subject (GELMAN *et al.*, 2020; MCELREATH, 2020; GELMAN *et*

al., 2013; KRUSCHKE, 2015).

BDA is a statistical framework concerned with practical methods for making inferences from data using probability models for quantities we observe and for quantities about which we wish to learn (GELMAN *et al.*, 2013). The simplified idea is that BDA takes a question in the form of a model and uses *logic* to produce an answer in the form of probability distributions (MCELREATH, 2020). This *logic* can be divided into two ideas: the first is that Bayesian inference is the reallocation of credibility (or formally, probability) across possibilities. The second one is that the possibilities over which we allocate probability are parameter values in meaningful mathematical models (KRUSCHKE, 2015).

The bayesian workflow includes the three major steps: model building, inference, and model checking/improvement (GELMAN *et al.*, 2020). These steps are not applied as a straight and linear process but rather as an interactive and incremental methodology. Usually, there are several model proposals, where we inferred from them to check their viability, which may demand some improvement. However, it is important to highlight that even if we linearly present them as follows, the back-and-forth in these steps is very common and highly recommended (GELMAN *et al.*, 2020; MCELREATH, 2020; KRUSCHKE, 2015).

4.1.1 Bayes in a Nutshell

Bayes 101. Bayesian inference is a statistical framework that allows us to update our subjective belief on the value of a variable of interest θ —or an *effect*— when faced with new data y. We start off by expressing our belief as a prior distribution $p(\theta)$ over the set Θ of possible values for θ. Then, we assume an observation model $p(y|\theta)$, and consequently a likelihood function that we can use to re-evaluate our opinion on θ. We refer to our updated belief as the *posterior* distribution and compute it using the Bayes' rule:

$$p(\theta|y) = \frac{p(\theta)p(y|\theta)}{\int_{\Theta} p(\theta,y)p(y)\,\mathrm{d}\theta}. \tag{4.1}$$

The first step in Bayesian modeling is to choose *likelihood* function $p(y|\theta)$, *i.e.*, an observation model. Choosing an appropriate likelihood requires analyzing the nature of the data y. For instance, if our observations are numbers of system failures in a given time interval, $p(\cdot|\theta)$ should have non-negative integer support — the most common choice for count being Poisson distribution $Pois(\lambda)$, governed by the parameter λ. If the data represent some natural or biological phenomena, commonly these cases are well represented (MCELREATH, 2020) by

using a Gaussian Distribution $\mathcal{N}(\mu, \sigma^2)$, with the mean μ and variance σ^2 parameters. Then, we must choose a prior. In the case of a *likelihood Pois*(λ), we must choose a prior $p(\lambda)$, which means the plausibility of the values that λ can assume before the data is observed. In case of a $\mathcal{N}(\mu, \sigma^2)$, we must choose priors $p(\mu)$ and $p(\sigma^2)$. Ideally, we can use previous analyzes and observations or specific domain knowledge to define *informative* priors. In cases where it is impossible to provide *informative* priors, we must at least ensure that the priors cover a reasonable value range or, conversely, rule out unusual value ranges as highly unlikely (FURIA *et al.*, 2021). These types of priors are called *weak* or *weakly informative*. We can test several *priors* and perform a sensitivity analysis to check which one best fits our data (GELMAN *et al.*, 2020). Generally, our hypotheses may not be represented by a single parameter θ in real-world scenarios. A hypothesis h takes the form of a mathematical model with several parameters, and we want to estimate them for a more precise representation of the hypotheses, considering the uncertainty about their values.

Computational methods. Once we have a model, the next step is making the inference. We must fit the model, which means we have to compute the posterior distribution $p(\theta|y)$. However, doing so analytically is often challenging since it requires computing the denominator of Bayes' rule, which is usually intractable. For simple cases with a small number of parameters, one can use grid or quadratic approximation to calculate the posterior (MCELREATH, 2020). Nonetheless, the weapon of choice for most Bayesians are Monte Carlo Markov Chain (MCMC) sampling methods, such as sequential Monte Carlo and Hamiltonian Monte Carlo (GELMAN *et al.*, 2020). In this chapter, we evaluate models that are computed using MCMC.

Hypothesis testing. The posterior distribution encapsulates all information we have gathered on θ, subjective or not, and we can use it to probe any hypothesis. For instance, we can evaluate the probability that $\theta > a$ taking the expected value of the indicator function $\mathbb{1}[\cdot > a]$, i.e.:

$$p(\theta > a) = \int_{\theta \in \Theta} \mathbb{1}[\theta > a] p(\theta|y) \, \mathrm{d}\theta.$$

Unfortunately, computing exact integrals over the posterior are often intractable. However, given a set of MCMC samples $\mathcal{S} = \{\theta^{(k=1)}\}^K$, we can approximate the expected value of any function g of θ as:

$$\mathbb{E}_{p(\theta|y)}[g(\theta)] = \int_{\theta \in \Theta} g(\theta) p(\theta|y) \, \mathrm{d}\theta \approx \frac{1}{K} \sum_{k=1}^{K} g\big(\theta^{(k)}\big).$$

Simulating novel data. We can easily simulate novel data with the posterior samples in our hands. We do so by sampling from:

$$\int_{\theta \in \Theta} p(y_\star, \theta | y) = p(y^\star | \theta) p(\theta | y) \, \mathrm{d}\theta,$$

which using MCMC samples resumes to repeating the following process: i) pick a sample $\theta^{(k)}$ from $\mathcal{S}$; ii) sample from our observation model conditioned on $\theta^{(k)}$, i.e., $p(y_\star | \theta^{(k)})$. Once the simulated data is drawn for a model that describes well our data's generative process, it should look similar to the observed data (GELMAN *et al.*, 2013), as any discrepancy between the sample data and the observed data indicates potential failings in the proposed model.

4.1.2 Hierarchical Models

The structure of data in a specific domain can indicate some relation or connection between the parameters of the model (GELMAN *et al.*, 2013). Consider the example of our selected dataset composed of 55 open source projects, all coming from the Apache Ecosystem, with all reports mined from Jira ITS. For instance, it might be reasonable to expect that these projects, coming from the same source, may present similar behavior in terms of types of bugs or the time to fix them, among other similarities. In this case, we can define in our model that the estimates of the parameter θ_i, representing the average time to fix a bug in a project i, are drawn from a prior distribution (conditioned by a parameter θ_0), also representing the average time to fix a bug, but considering *all* projects behavior.

The advantage of this kind of approach, called Hierarchical or Multi-level models, is that they are less inclined to underfit or overfit the data when compared to single-level models, dealing better with the imbalance in sampling and better modeling between variance among groups and individuals (MCELREATH, 2020). From an intuitive point of view, this mechanism allows for transferring information between different projects. This also will enable projects with fewer data to borrow strength from inferences in more mature projects. Besides allowing us to estimate the parameters θ of each project, hierarchical modeling also provides us with a distribution over the global parameter θ_0 — in our case, the global average BFT —, which is more suitable for generalization results.

4.2 Selected Features

We choose to analyze three bug reports features: i) links; ii) priority; iii) code-churn. This analysis aims to verify how these features relate to the bug resolution time. Given a project, for each feature, we create groups of reports based on the specific criteria of the selected feature values:

- For 'links', we split the data into two groups: the group of reports with links (R_{wl}) and the group with no links (R_{nl}). As presented in Subsection 3.1.3, there are several scenarios of links in the reports. However, for this first round of analysis, we choose only to consider the existence or not of *some* link.
- For 'priority', we split the data into three groups: the group of reports with trivial-minor priority (low priority, R_{lp}), the group with major priority (medium priority, R_{mp}) and the group with critical-blocker priority (high priority, R_{hp}). This group of the lower and higher priorities values is justified for two major reasons: the Cassandra project uses three levels of priority (low, normal, and urgent), and some smaller projects do not contain examples of reports with all priorities. With this grouping strategy, we can deal with all the projects simultaneously.
- For 'code-churn', we define it as the sum of added and removed lines in this context. We split the data into two groups: reports with higher code-churn values (R_{hcc}) and reports with lower code-churn size (R_{lcc}). Given a project, the threshold to split both groups is the median code-churn of its bug reports.

We justify the interest in studying the relation between bug fix time with each one of the features as follows. The relation (links) between issue reports seems to be overlooked by papers that study bug reports. For instance, with a quick search for papers with keywords such as 'bug reports', 'links', 'relationship', 'Jira reports', we only were able to find three papers that deal explicitly with relations in bug reports (BORG *et al.*, 2013; TOMOVA *et al.*, 2018; THOMPSON *et al.*, 2016). Also, none of the proposals that use machine learning techniques to estimate fixing time consider the relationship between reports as features (See Section 3.5). Links can represent several types of relation, as presented in Fig. 5. It is reasonable to believe that a blocked report will only be fixed after the blocker report is resolved, implying some interference of a report over another. Other types of relationships, as duplicated, are also an indication that fixing one report can impact considerably other ones.

On the other hand, the priority is an objective of the study in several papers that deal

with similar data and are almost used in all predictive models as features. However, it is never clear how a bug report priority is directly related to a bug fixing time. An argument that the priority is a measure of importance or urgency, hence asks for more attention and rapid responses. However, priority carries no information about the complexity of the bug (i.e., a minor bug may be more complex than a simple but blocker bug). With this analysis, we intend to bring some light to the matter.

Code-churn is a widespread metric in software engineering research. However, it is post-bug fix information: it is only known after the bug resolution. So, how does this information on the relation between code churn size and bug fixing time improve the bug fixing process? We argue that once the bug is located in a class, function, or file, one could use prior information about the code churn values in this specific bug location to estimate (along with the report information) the time to fix the bug.

4.3 Modeling Process and Models Description

The analysis starts with two proposed models as hypotheses to explain the generative data process. Given a feature, we fit both models for each one of its groups. After that, we compare the adverse groups' μ posterior distribution to draw our conclusions. For instance, for the 'links' analysis, we first fit a model using R_{nl} data and then another model using the R_{wl} data. Then we use each model's μ posterior distribution to verify the difference between both groups.

We first define some sets, distributions, and variables that we use to describe the models:

- *days* / d: The time to fix the bugs in days, as non-negative real numbers. For all models, we considerer $log(days) \sim \mathcal{N}(\mu, \sigma^2)$, as *days* can not assume negative numbers.
- $\mathcal{G}$: the groups of bug fixing time in *days*. The groups of data are $\mathcal{G} = \{R_{nl}, R_{wl}, R_{lp}, R_{mp}, R_{hp}, R_{lcc}, R_{hcc}\}$, as presented in subsection 4.2.
- $\mathcal{P}$: The set of all projects. $\mathcal{P} = \{p_1, ..., p_{55}\}$, each p_i being one of the projects presented in Table 1.
- $\mathcal{N} \sim (\mu, \sigma^2)$: The Normal distribution, defined the parameters mean μ and variance σ^2
- Inv-Gamma(α, β) / Γ^{-1}: The Inverse Gamma distribution, defined by parameters α and β. Usually, the Inverse Gamma is used as prior for the variance in BDA.

The first model is a specific model, where we fit using only one project data at a time. We use a weakly informative prior for all parameters. The following model, described in as an

equation 4.2 and graphically in Fig. 15 is called ‘specific-model’.

$$
\begin{aligned}
&\mu \sim \mathcal{N}(0,2), \\
&\sigma^2 \sim \text{Inv-Gamma}(3,3), \\
&\log(\text{days}) \sim \mathcal{N}\left(\mu, \sigma^2\right).
\end{aligned}
\tag{4.2}
$$

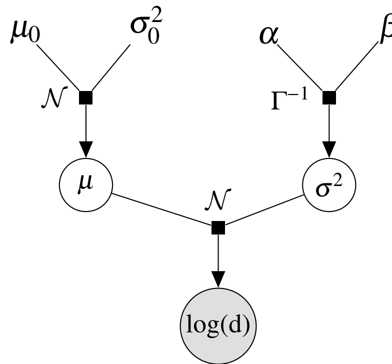


Figure 15 – Specific Model representation. Weakly informative priors are used for parameters μ and σ^2 with values $\mu_0 = 0$; $\sigma^2 = 2$; $\alpha = 3$; $\beta = 3$.

Using this model, we fit a total of 385 models: 110 for links (for each of the 55 projects, we fit a model using R_{wl} data and another using R_{nl} data), 110 for code-churn (for each of the 55 projects, we fit a model using R_{lcc} data and another using R_{hcc} data) and 165 for priority (for each of the 55 projects, we fit a model using R_{lp} data, another using R_{mp} and another using R_{hp}).

The second model is a Hierarchical Model (HM), where we fit all projects data at once. We also use a weakly informative prior for all proposed models. We have a μ_0 representing the parameter to estimate for all projects population, while we have one μ_i to describe each project p_i. The following model is called ’HM-AP,’ and it is described in equation 4.3 and graphically in Fig. 16.

$$
\begin{aligned}
&\mu_0 \sim \mathcal{N}(0,2),\ \sigma_0^2 \sim \text{Inv-Gamma}(3,3), \\
&\sigma_i^2 \sim \text{Inv-Gamma}(3,3), \forall p_i \in \mathcal{P}, \\
&\mu_i \sim \mathcal{N}\left(\mu_0, \sigma_0^2\right), \forall p_i \in \mathcal{P}, \\
&\log(\text{days}_i) \sim \mathcal{N}\left(\mu_i, \sigma_i^2\right), \forall i \in \mathcal{P}.
\end{aligned}
\tag{4.3}
$$

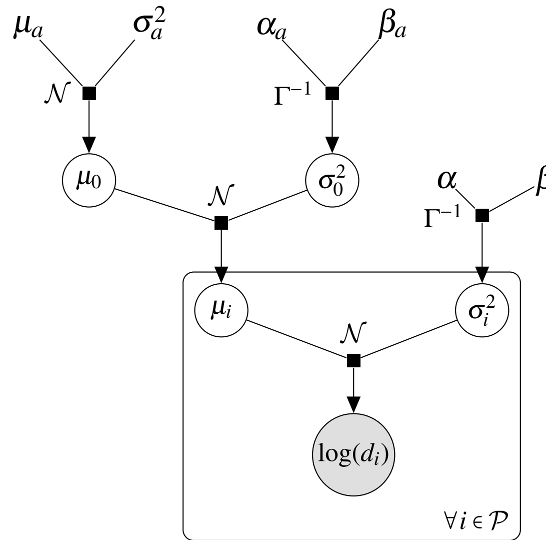


Figure 16 – Hierarchical Model - HM-AP representation. Weakly informative priors are used for paramters μ_0, μ_i, σ_0^2, and σ_i^2. Values are $\mu_a = 0$; $\sigma_a^2 = 2$; $\alpha_a = \alpha = 3$; $\beta_a = \beta = 3$.

Using the 'HM-AP', we fit a total of seven models: two for links (one using R_{wl} data of all projects and another using R_{nl} data of all projects), two for code-churn (one using R_{lcc} and another using R_{gcc} data of all projects) and three for priority (same logic as previous, R_{lp}, R_{mp}, and R_{hp} data of all projects). The proposed hierarchical model intends to capture the global bug report behavior based on the particular data of each project. 'HM-AP' assumes that there is no other similarity aspect between the projects besides they all are bug reports.

4.4 Results

With all models defined, we use Stan, specifically PyStan[1] which is a Python interface to Stan, a package for Bayesian inference. Stan is a state-of-the-art platform for statistical modeling and high-performance statistical computation. The computation goes as presented in 4.1: we compute the posterior distribution $p(\mu, \sigma^2)$ using MCMC for each model, given the data presented in $\mathcal{G}$. After that, we compare the opposite μ posterior distributions for each feature group using five summarizations: the Maximum a Posteriori μ_{MAP} estimator (the most plausible value for the estimator μ); the Lower (CI_L) and Upper (CI_U) value of the 95% Confidence Interval (CI, also known as Uncertainty, Credible, or Compatibility Interval); the probability of a

[1] https://pystan.readthedocs.io/en/latest/

group having a greater average fixing-time than the other (4.4); and the expected value of the difference between both bug-fixing time groups (4.5), as presented in the following equations

$$f_1(a,b): 1 \text{ if } a > b, \text{ 0 otherwise,}$$
$$E[f_1(a,b)] = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f_1(a,b)p(a)p(b)\,\mathrm{d}a\,\mathrm{d}b. \tag{4.4}$$

$$f_2(a,b): a - b,$$
$$E[f_2(a,b)] = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f_2(a,b)p(a)p(b)\,\mathrm{d}a\,\mathrm{d}b. \tag{4.5}$$

We present the results visually through the μ posterior distributions for each data group, the μ_{MAP} represented by a dotted line, and the Confidence Interval (CI) by the filled area under the curve. We also present the numeric values with an associated table for each proposed model, the summarization tables, along with the values obtained using the equations 4.4 and 4.5.

The results are grouped by feature and similarly presented in the following subsections. First, we show the results using the 'specific-model' of only four projects, due to size constrain, but also to offer some divergent scenarios regarding the possible conclusions about the difference between the groups of features. The complete posterior distribution visualisations and summarization tables for all 55 projects can be found in the Appendix B.

4.4.1 Links

The Fig. 17 shows four projects 'specific-models' marginal μ posterior distributions, while Table 16 the distribution summarization.

The conclusions differ based on the project we analyze. For Derby project results, the difference between both groups is evident, with the reports with links taking more time to be fixed than those with no links. In Hadoop Map/Reduce, we also see a similar behavior but with some overlap of both distributions. In Oozie, we see an inverse behavior: reports with no links present higher bug fixing time than those with links. Finally, the Lang project shows no difference between both groups. While most projects present a behavior similar to Derby and Hadoop Mapreduce (see appendix B), it is hard to conclude the real impact of links in the report bug fixing time taking each project individually. The results give us a general picture of each project's behavior but do not help us to verify a bug reports global behavior.

As the project's analysis does not help to answer our research question, this justifies using hierarchical models to summarize the population's behavior of bug reports. We present

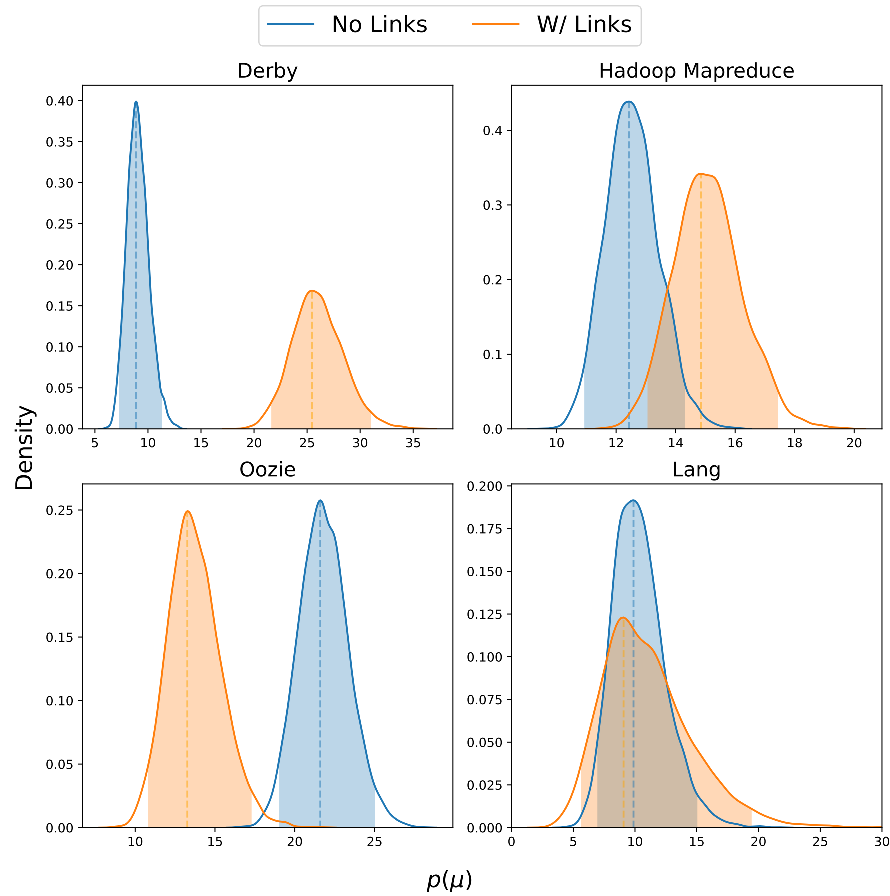


Figure 17 – μ posterior distributions - ‘specific-models’, ‘links’ results. The average bug fixing time of reports with no links vs. those with links. The conclusions diverge depending on the selected project.

Table 16 – μ posterior distribution summary, ‘specific-models’, ‘links’ results.

No Links (a)

Project	CI_L	CI_U	$\hat{\mu}_{MAP}$	E[F1(a,b)]	E[F2(a,b)]
Derby	7.24	11.35	8.86	-16.93	0.00
Lang	6.87	15.10	9.88	-0.68	0.46
Mapreduce	10.92	14.35	12.44	-2.55	0.04
Oozie	18.97	25.04	21.60	8.07	1.00

W/ Links (b)

Project	CI_L	CI_U	$\hat{\mu}_{MAP}$	E[F1(b,a)]	E[F2(b,a)]
Derby	21.55	31.08	25.46	16.93	1.00
Lang	5.52	19.45	9.08	0.68	0.54
Mapreduce	13.01	17.45	14.85	2.55	0.96
Oozie	10.79	17.29	13.27	-8.07	0.00

the μ_0 posterior distributions obtained using the 'HM-AP' in Fig. 18 and the summarization in Table 17.

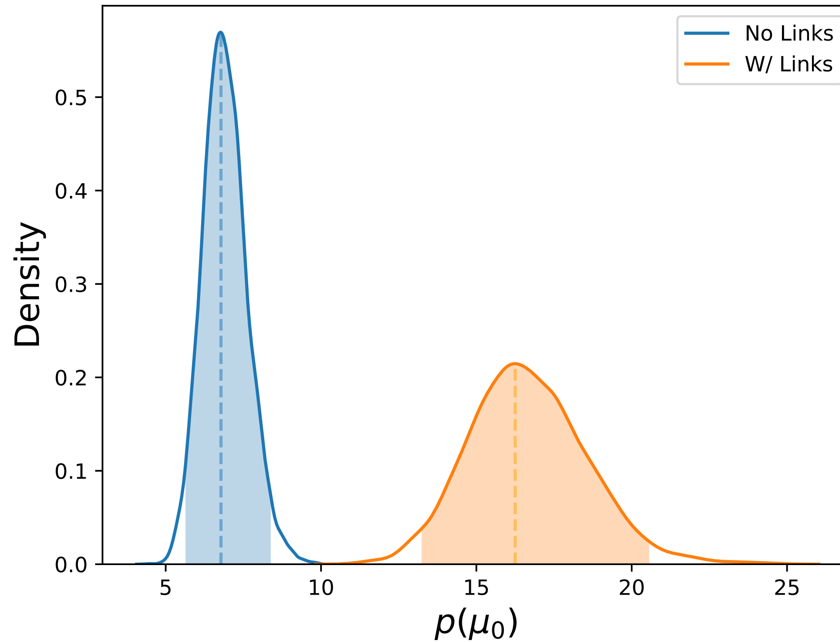


Figure 18 – μ_0 posterior distributions - 'HM-AP', 'links' results. The average bug fixing time of all reports with no links vs. all reports with links. The results indicates that bugs with links take more time to be fixed than their counterparts.

Table 17 – μ_0 posterior distribution summary. 'HM-AP', 'links' results.

No Links (a)

CI_L	CI_U	$\mu_{0_{MAP}}$	E[F1(a,b)]	E[F2(a,b)]
5.63	8.41	6.78	-9.76	0.00

W/ Links (b)

CI_L	CI_U	$\mu_{0_{MAP}}$	E[F1(b,a)]	E[F2(b,a)]
13.20	20.59	16.25	9.76	1.00

The results show a significant difference between both average bug fixing times, where reports with links (group 'a') need more time to be fixed than reports with no link (group 'b'). The expected difference between both groups is 9.76 days, suggesting that bugs with links tend to take 2.4 more times to be fixed than those with no links. The probability of group 'b' is greater than group 'a' is 1. In this context, we answer our **RQ1**: *considering the posterior distribution of μ_0 for groups of reports with links and no links, along with their summarization, we gather evidence that the bugs with links tend to need approximately 2.4 times to be fixed when*

compared to reports with no links.

4.4.2 Priority

The Fig. 19 shows four projects 'specific-models' μ posterior distributions, while Table 18 the distributions summarization. Once again, as presented in the results for links, we selected four different scenarios of possible conclusions.

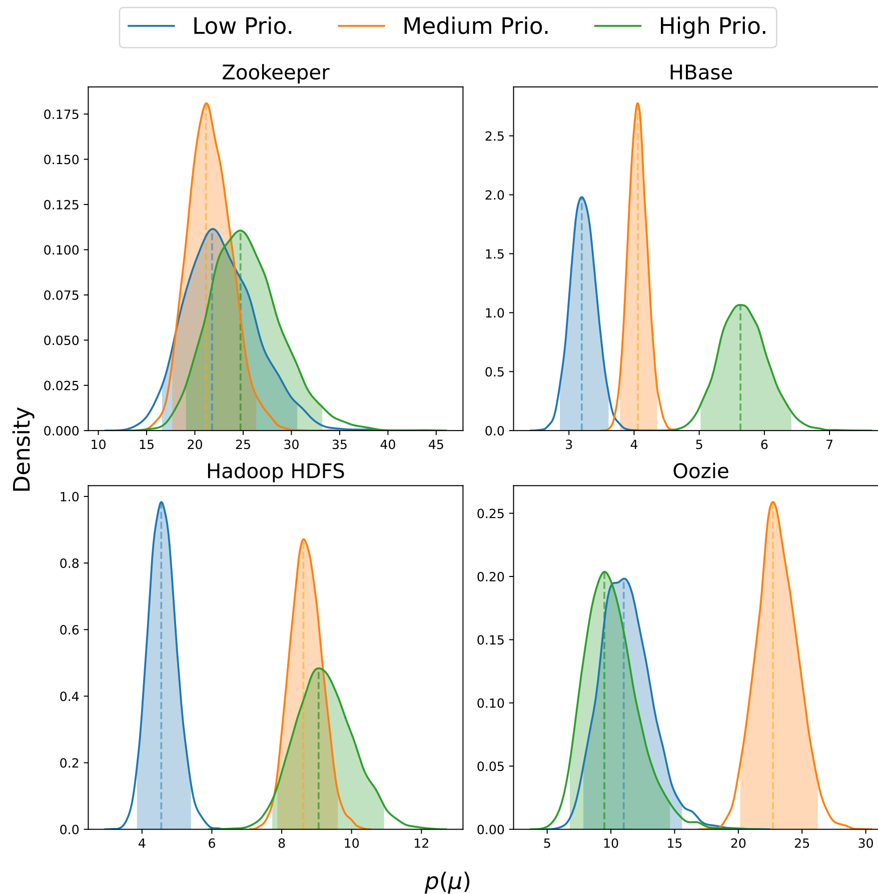


Figure 19 – μ posterior distributions - 'specific-models', 'priority' results. The average bug fixing time of three reports priority levels. The conclusions diverge depending on the selected project.

Depending on the selected project, the conclusions diverge. For Zookeeper, the distributions are majority overlapped, indicating little significant difference between the three groups. The data from HBase presents a very distinct behavior for each group, with the order of bug-fixing time being low, medium, and high priority. For both HDFS and Oozie, only one group presents a more distinct behavior when compared with the other two. In HDFS, bugs with low priority take less time than bugs with medium and high priority, both presenting very similar estimators values for μ. With Ozzie, we also notice a similarity between low and high priority

Table 18 – μ posterior distribution summary, 'specific-models', 'priority' results.

Low Priority (a)

Project	CI_L	CI_U	μ_{MAP}	E[F1(a,b)]	E[F1(a,c)]	E[F2(a,b)]	E[F2(a,c)]
HBase	2.85	3.61	3.20	-0.84	-2.46	0.00	0.00
HDFS	3.85	5.42	4.55	-4.11	-4.65	0.00	0.00
Oozie	7.78	15.64	11.01	-11.84	1.18	0.00	0.67
Zookeeper	16.45	30.69	21.79	1.21	-2.51	0.60	0.31

Medium Priority (b)

Project	CI_L	CI_U	μ_{MAP}	E[F1(b,a)]	E[F1(b,c)]	E[F2(b,a)]	E[F2(b,c)]
HBase	3.78	4.36	4.06	0.84	-1.62	1.00	0.00
HDFS	7.87	9.62	8.62	4.11	-0.54	1.00	0.29
Oozie	20.11	26.27	22.72	11.84	13.01	1.00	1.00
Zookeeper	17.64	26.38	21.16	-1.21	-3.73	0.40	0.19

High Priority (c)

Project	CI_L	CI_U	μ_{MAP}	E[F1(c,a)]	E[F1(c,b)]	E[F2(c,a)]	E[F2(c,b)]
Hbase	5.01	6.43	5.63	2.46	1.62	1.00	1.00
HDFS	7.72	10.93	9.06	4.65	0.54	1.00	0.71
Oozier	6.71	14.64	9.49	-1.18	-13.01	0.33	0.00
Zookeeper	18.97	33.31	24.73	2.51	3.73	0.69	0.80

bugs, while reports with medium priority take more fixing time. As presented in the results with links, it is hard to conclude the real impact of priority in the report bug fixing time, taking each project individually.

We use the hierarchical 'HM-AP' to draw our conclusions for the priority group. The μ_0 bug reports population posterior distributions is presented in Figure 20 and the summarization in Table 19.

Table 19 – μ_0 posterior distribution summary, 'HM-AP' model, 'priority' results.

Low Priority (a)

CI_L	CI_U	$\hat{\mu}_{MAP}$	E[F1(a,b)]	E[F1(a,c)]	E[F2(a,b)]	E[F2(a,c)]
6.53	10.04	8.03	-0.37	-0.12	0.39	0.47

Medium Priority (b)

CI_L	CI_U	$\hat{\mu}_{MAP}$	E[F1(b,a)]	E[F1(b,c)]	E[F2(b,a)]	E[F2](b,c)
6.93	10.50	8.31	0.37	0.25	0.61	0.58

High Priority (c)

CI_L	CI_U	$\hat{\mu}_{MAP}$	E[F1(c,a)]	E[F1(c,b)]	E[F2(c,a)]	E[F2(c,b)]
6.41	10.57	7.92	0.12	-0.25	0.53	0.42

The results suggest that the difference between the groups of reports with distinct

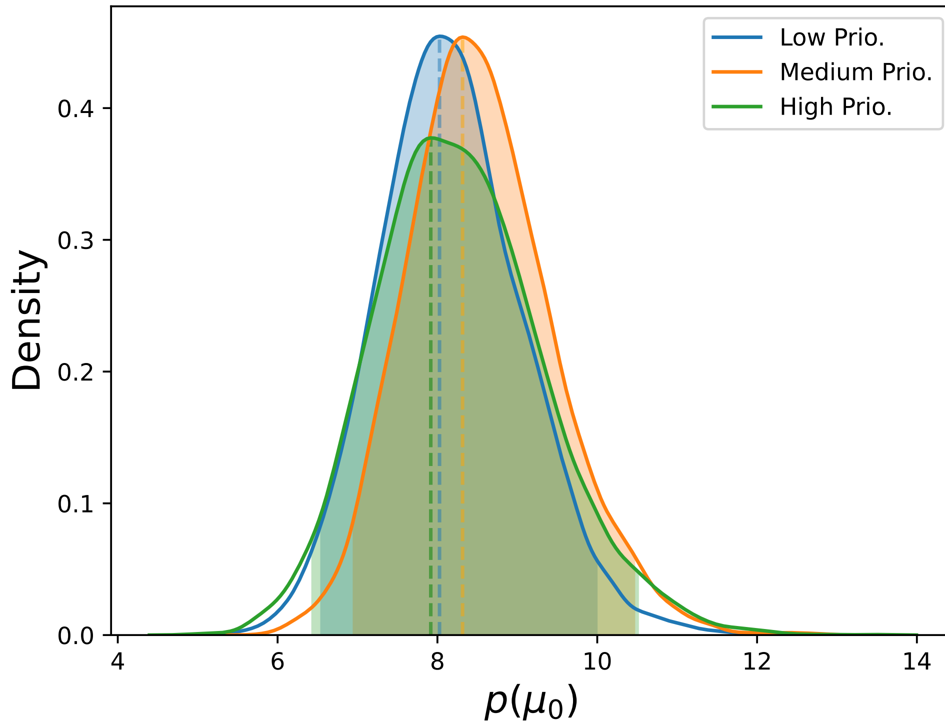


Figure 20 – μ_0 posterior distributions - 'HM-AP' model, 'priority' results. The overlap and similarities between the posterior distributions indicate a low priority influence on the BFT.

priority is unclear. For example, although there are different levels of uncertainty interval across the μ distributions, the μ_{MAP} values are very similar. The average difference time between the groups (E[F1] values) is small. In this context, we answer the **RQ2**: *considering the posterior distribution of* μ *for groups of reports with low, medium, and high priority, along with their summarization, we gather evidence that the bug's priorities do not have a significant impact in the bug-fixing time*.

4.4.3 Code Churn

The Fig. 21 shows four projects 'specific-models' marginal μ posterior distributions, using the code-churn data groups, while Table 20 the distributions summarization. Once again, as presented in the results for previous features, we selected four different scenarious of possible conclusions.

The code-churn results present similar behavior as presented in the links results. For example, Flink and Crunch results show that patches with higher code-churn take more time to be fixed than those with smaller code-churn sizes. However, in Flink, the difference is evident, while Crunch presents a significant overlap. Maven project presents an inverse behavior, with

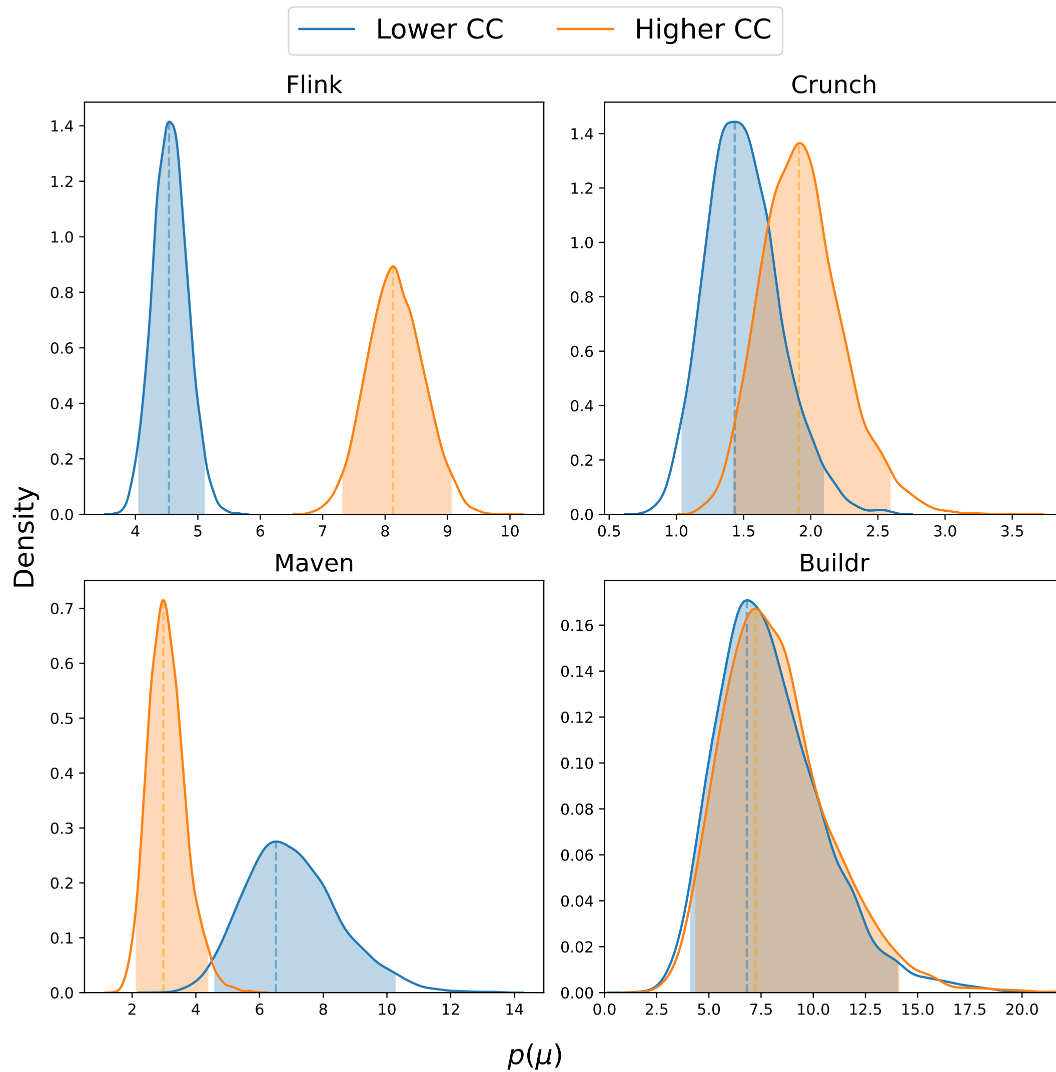


Figure 21 – μ posterior distributions - 'specific-models', 'code-churn' results. The average BFT of bugs patches with higher vs. lower code-churn values. The conclusions diverge depending on the selected project.

Table 20 – μ posterior distribution summary, 'specific-models', 'code-churn' results.

Lower Code-Churn (a)

Project	CI_L	CI_U	μ_{MAP}	E[F1(a,b)]	E[F2(a,b)]
Buildr	4.10	14.08	6.82	-0.26	0.47
Crunch	1.03	2.11	1.43	-0.44	0.14
Flink	4.04	5.12	4.54	-3.60	0.00
Maven	4.56	10.30	6.52	3.95	1.00

Higher Code-Churn (b)

Project	CI_L	CI_U	μ_{MAP}	E[F1(a,b)]	E[F2(a,b)]
Buildr	4.24	14.28	7.24	0.26	0.53
Crunch	1.42	2.60	1.91	0.44	0.86
Flink	7.31	9.07	8.13	3.60	1.00
Maven	2.10	4.40	2.98	-3.95	0.00

patches with higher values of code-churn taking less time to be fixed than the smaller ones. In Buildr, the values for estimator μ are almost identical. Once again, the particular nature of each project shows dissonant conclusions.

We fit another 'HM-AP' to draw our conclusions for code-churn group. The marginal μ_0 bug reports population posterior distributions of is presented in Fig. 22 and the summarization in Table 21.

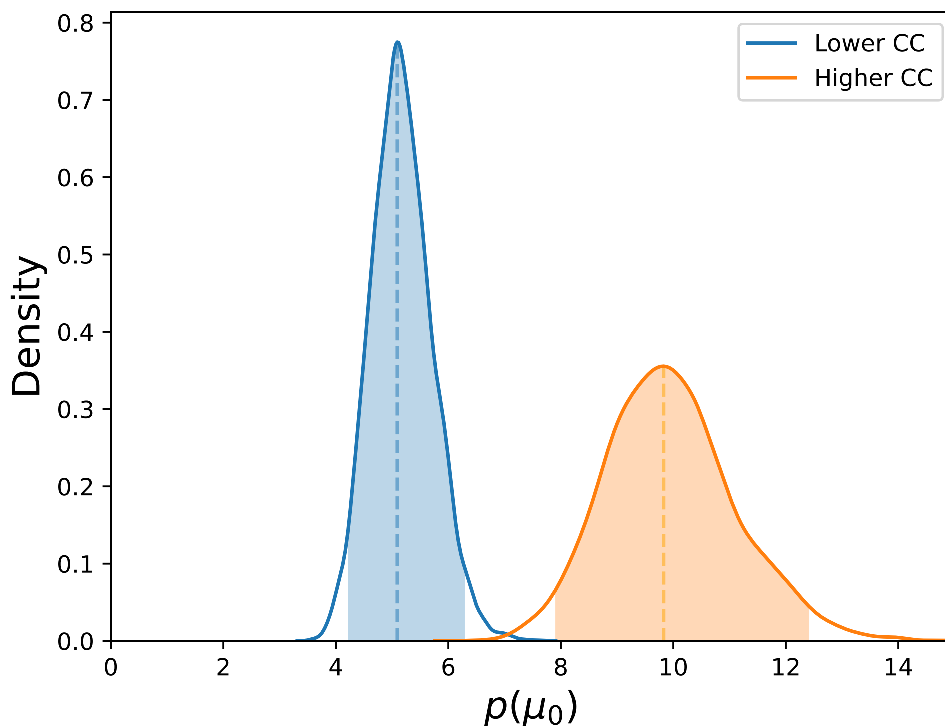


Figure 22 – μ_0 posterior distributions - 'HM-AP', 'code-churn' results. The average BFT of all reports with lower vs. all reports with higher code-churn values. The results indicate that bugs with higher code-churn patches values demand time to be fixed than their counterparts.

Table 21 – μ_0 posterior distribution summary, 'HM-AP' model, 'code-churn' results.

Lower Code-Churn

Model	CI_L	CI_U	$\mu_{0_{MAP}}$	E[F1(a,b)]	E[F2(a,b)]
All Projects	4.20	6.31	5.09	-4.78	0.00

Higher Code-Churn

Model	CI_L	CI_U	$\mu_{0_{MAP}}$	E[F1(b,a)]	E[F1(b,a)]
All Projects	7.89	12.42	9.83	4.78	1.00

The results show a significant difference between both average bug fixing times, where reports patches with higher code-churn values (group 'a') need more time to be fixed than

reports patches with smaller code-churn values (group ‘b’). The expected difference between both groups is 4.78 days, suggesting that bugs of the group ‘a’ take double the time to be fixed than those of group ‘b’. The probability of group ‘b’ is greater than group ‘a’ is 1. In this context, we answer our **RQ3**: *considering the posterior distribution of μ_0 for groups of reports with higher and smaller code-churn values, along their summarization, we gather evidence that the bugs reports paths with greater code-churn values tend to need approximately 5 more days (the double of time) to fixed compared to bugs reports paths with smaller code-churn values.*

4.5 Exploring different Hierarchical Models

The ‘HM-AP’ model considers that all projects come from the same distribution with mean μ_0. However, hierarchical models allow us to consider more information regarding other data particularities, for instance, the existence of other subgroups in the data. A straightforward example in the case of our dataset is the projects’ categories. Based on the idea that projects of the same category are more similar in terms of bug-fix time and types of bugs, we can add this assumption to the model composition.

The category is only one example of how we can arrange the data into a tree-like structure that models similarities between the projects. However, one can propose another hypothesis, as long they are appropriate to the data domain. This section explores three different ways to structure the data: category, maturity (years of existence), and the number of reported bugs. This subsection’s objective is to show the hierarchical models’ flexibility and how we can explore different data generation processes.

This subsection follows a similar structure of previous Subsection 4.3 and 4.4. We first describe the new groups of data, the models, and the results (the posterior distribution and summarization table) obtained by fitting the models with ’links’ data. We also perform the same analysis with ’priority’ and ’code-churn’ data, which can be found in the Appendix C.

First, we define the new subgroups of data:

- $\mathcal{C}$: The set of projects grouped by category. $\mathcal{C} = \{C_1, \ldots, C_9\}$, each C_i representing a set of projects in the same category, as presented in Table 1.
- $\mathcal{Y}$: The set of projects grouped by maturity. $\mathcal{Y} = \{Y_{2009}, Y_{2010-2012}, Y_{2013-2015}\}$, three groups of projects defined by the year that contains the first bug report recorded in the Jira ITS. The groups are:
  - Y_{2009}: {‘Lang’, ‘Zookeeper’, ‘Nutch’, ‘Dirmina’, ‘Vysper’, ‘Hadoop Mapreduce’,

'Tap5', 'Dirkrb', 'SSHD', 'Compress', 'Hadoop HDFS', 'Solr', 'Myfaces', 'MRM', 'Codec', 'MNG', 'Derby', 'Ivy', 'Lucene', 'Camel', 'Tika', 'Mahout', 'Cassandra', 'Hadoop Core', 'Math', 'VCL', 'Hbase', 'Hive', 'IO', 'FTPserver', 'Collections', 'Buildr', 'Openjpa', 'WW'};

- $Y_{2010-2012}$: {'Yarn', 'Log4j2', 'Libcloud', 'Syncope', 'Giraph', 'Oozie', 'Madlib', 'Tomee', 'Crunch', 'Helix', 'Kafka', 'Isis', 'Spark', 'Mesos'};
- $Y_{2013-2015}$: {'Flink', 'Systemml', 'Jclouds', 'FC', 'Storm', 'Phoenix', 'Ignite};

- $\mathcal{Q}$: The set of projects grouped by size (number of bug reports), where the groups division is defined by quartils. $\mathcal{Q} = \left\{Q_{[0,q1]}, Q_{]q1,q2]}, Q_{]q2,q3]}, Q_{]q3,100]}\right\}$, each index i representing the interval where each project is in the quartils division.

Considering one of these groups at a time, we can create three new hierarchical models. Using the same design from the 'HM-AP', we add another level to the model considering new data clusters. Let $\mathcal{G}$ be a generic symbol that can be replaced by any of the groups $\mathcal{C}$, $\mathcal{Y}$, or $\mathcal{Q}$. The new models set a new level regarding group $\mathcal{G}$ parameters ($\mu_G, \forall G \in \mathcal{G}$), between parameters about the population (μ_0) and specific projects (μ_i). All models follow the same structure presented in (4.6) and Figure 23:

$$
\begin{aligned}
&\mu_0 \sim \mathcal{N}(0,2),\ \sigma_0^2 \sim \text{Inv-Gamma}(3,3)\\
&\sigma_G^2 \sim \text{Inv-Gamma}(3,3),\ \forall G \in \mathcal{G}\\
&\sigma_i^2 \sim \text{Inv-Gamma}(3,3),\ \forall p_i \in \mathcal{P}\\
&\mu_G \sim \mathcal{N}\left(\mu_0, \sigma_0^2\right),\ \forall G \in \mathcal{G}\\
&\mu_i \sim \mathcal{N}\left(\mu_G, \sigma_G^2\right)\ \forall p_i \in \mathcal{P},\ \forall G \in \mathcal{G} \mid p_i \in G\\
&\log(\text{days}_i) \sim \mathcal{N}\left(\mu_i, \sigma_i^2\right),\ \forall p_i \in \mathcal{P}
\end{aligned}
\tag{4.6}
$$

All proposed hierarchical models intend to capture the global bug report behavior based on the detailed data of each project. However, each one evaluates a different hypothesis on the generative data process. 'HM-AP' assumes that there is no other similarity aspect between the projects besides they all are bug reports. The other HMs add another layer of information, suggesting clusters of similarity that can help the modeling data process. The 'HM-$\mathcal{C}$' is a natural suggestion of another information that aggregates groups of projects. For example, one hypothesis to explain why category would matter in terms of bug reports fixing time is, as they deal with the same domain, their bugs can be more similar, hence taking an approximated time to be fixed. The 'HM-$\mathcal{Q}$' model can represent a hypothesis of more complex projects that presents

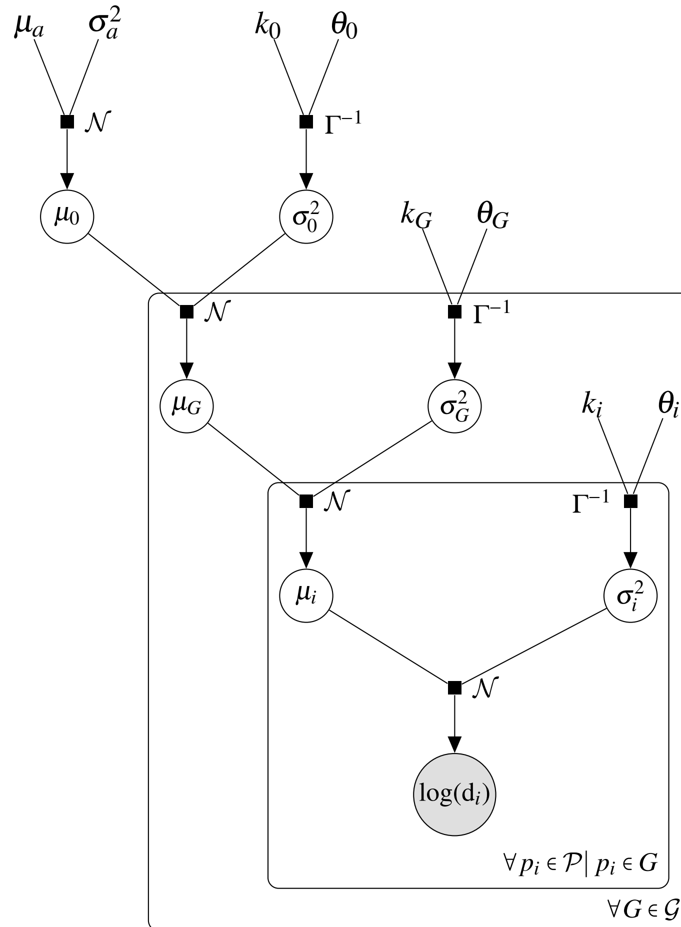


Figure 23 – Hierarchical Model 'HM-$\mathcal{G}$' - Graph representation for projects considering subgroups $\mathcal{G}$ of projects, where $\mathcal{G}$ can be replaced by $\mathcal{C}$, $\mathcal{Y}$, or $\mathcal{Q}$.

more bugs and can have a similar process to triage that would impact the fixing time. Finally, the 'HM-$\mathcal{Y}$' would verify the viability of older projects by providing similar, more robust conduct regarding the bug fixing process.

Figure 24 shows the μ_0 posterior distribution of each alternative hierarchical model. Table 22 presents the summarization of each model.

All models suggest that reports with links have a greater fixing time than those without links. However, there is more uncertainty about the differences between the values of μ_0 of both reports groups, especially when compared with the results using the model 'HM-AP'. The result obtained using model HM-$\mathcal{C}$ shows that the distinction between both groups is more evident when compared with models HM-$\mathcal{Q}$ and HM-$\mathcal{Y}$, which present a more prominent overlapping between the distributions. However, the E[F2] function values indicate a high probability of reports with links demanding more time to be fixed.

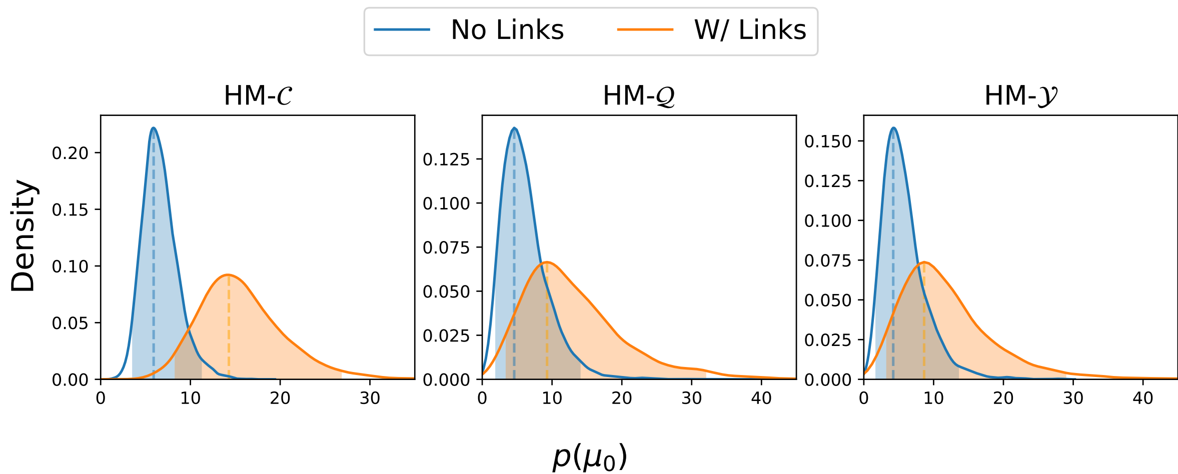


Figure 24 – μ_0 posterior distributions - 'HM-$\mathcal{G}$', 'links' results. The three models indicate that bugs with links demand higher BFT, but with less confidence than the results obtained with model 'HM-AP'.

Table 22 – μ_0 posterior distributions summaries, 'HM-$\mathcal{G}$' models, 'links' results.

No Links (a)

Model	CI_L	CI_U	μ_{MAP}	E[F1(a,b)]	E[F2(a,b)]
HM-$\mathcal{C}$	3.49	11.36	5.90	-9.11	0.03
HM-$\mathcal{Q}$	1.60	14.32	4.58	-7.08	0.18
HM-$\mathcal{Y}$	1.56	13.76	4.22	-6.14	0.19

W/ Links (b)

Model	CI_L	CI_U	μ_{MAP}	E[F1(b,a)]	E[F2(b,a)]
HM-$\mathcal{C}$	8.03	26.94	14.28	9.11	0.97
HM-$\mathcal{Q}$	2.98	32.40	9.29	7.08	0.82
HM-$\mathcal{Y}$	2.72	29.59	8.64	6.14	0.81

4.6 Discussion

This Chapter presents an analysis of the interplay between the three bug report features - links, priority, and bug-fixing code churn size - and the bug-fixing time. We use the BDA workflow on the data of 55 open source projects from the Apache ecosystem. We propose two models as the hypothesis for the data generation process. Based on the obtained results, we gather evidence that priority plays no role in the bug fix time. In contrast, bug reports with higher values of code churn and bugs reports that relate to other bugs need at least, on average, double the time to be fixed compared to their counterparts.

Regarding the results, we highlight a few points. First, the relationship between bug reports seems to be overlooked, which appears to be wasted potential for deeper analysis and BFT predictive models. We look to use Graph Neural Network (ZHOU *et al.*, 2018) to verify how these relations can improve the state-of-art estimation results in future works. The results

regarding code-churn can also be an exciting addition to BFT estimation when used with bug localization strategies. To conclude the analysis of the results, the priority not having much evidence of being impactful on the BFT may not discard it entirely from being used in predictive models, as it can be related to other features. The priority seems to provide some contextual information about the situation of an open bug compared to other ones. Priority may play a role when the context of the specific report is known, which is rarely the case in several papers. For instance, a newly reported high-priority bug (blocker, critical) can take more time to be fixed if it competes for resources with several other high-priority bugs. The same report could be fixed early in a scenario of several low-priority bugs. Also, it can be related to other responses in the platform as response-time from other developers, the number of comments and watches, or the time to review proposed patches. Exploring these hypotheses seems to be an exciting path to understanding the priority role in the bug fixing process.

4.7 Threats to the Validity

The threats to the validity of our investigation are discussed using the four threats classification (conclusion, construct, internal, and external validity) presented by WOHLIN *et al.* (WOHLIN *et al.*, 2012).

Conclusion Validity the main threats to this validity concern the choices we made in the modeling process: the weakly prior and the likelihood function. Regarding the first choice, Bayesian statistics results benefit from using more representative prior, ideally from a different data source (i.e., bug reports from other projects or posterior from previous analysis). However, even if we don't use data from other projects, we perform a sensitivity analysis to provide a reasonable prior based on earlier studies that deal with bug-fixing time. For example, some studies from Weiss *et al.* (2007), Akbarinasaji *et al.* (2018) suggest that bugs are generally fixed in a few days, while other studies show that some bugs can take months or even years to be considered fixed (SAHA *et al.*, 2014). We provide a broad enough prior distribution to consider these cases. The expected bug-fixing time value is close to 2~3 days (most common cases) but also allows the model to contemplate instances with hundreds of days (although these cases are less plausible, represented in the prior). Appendix C provides two sensitivity analyses to cover the two threats. We show that the selected parameters, distributions, and likelihood functions are appropriate: a predictive analysis of the prior and an analysis of how well a log-normal fits the log-BFT of most projects.

Internal Validity The existence of other features that can be highly correlated with the analyzed features and that can be the *actual* causal effect of the bug-fixing time. For instance, we show that reports with links present higher BFT. However, our analysis does not consider other features (*e.g.*, the number of comments and the existence of attached patches) that can be highly correlated with the presence of links and are the actual cause of higher BFT. We argue that the number of analyzed projects and bug reports — more than 70.000 reports from 55 projects — mitigate the chances of these correlations propagating thought all projects.

Construct Validity In the 'links' analysis, we had to ignore the types of relations between reports, only considering the *existence* of a link. This simplifies the investigation because we are not able to indicate which type of link really impacts and how much it impacts the BFT. However, we had to perform this simplification due to the size of a few projects, as some of them do not have enough data to perform this level of type-of-links groups granularity.

External Validity All projects are open-source from the Apache ecosystem, indicating some source of low generalization capability. However, we argue that the sample contains 55 projects from nine categories (big-data, database, machine learning, and library, to cite some), with different maturity levels, some of them dated from 2002 and others from 2018. We mine the dataset, providing diversity, which allows us to generalize the results with more certainty.

4.8 Related Works

Hooimeijer e Weimer (2007) discuss the process of modeling bug reports quality. The authors present a descriptive model of bug report quality based on 27,000 bug reports for the Mozilla Firefox projects. The analysis shows that the presence of an attachment tends to lead to higher values of bug-fixing time, while the comment count suggests that bugs that receive more attention get fixed faster. The self-reported severity (value given in the bug report creation) also plays a role in bug fixing.

Zimmermann *et al.* (2010) investigates the quality of bug reports from the perspective of developers. To find out which features and elements matter the most, they asked several developers from Apache, Eclipse, and Mozilla projects to perform two tasks: i) a survey on bug reports important information and ii) rate the quality of bug reports on a five-point Likert scale (from very poor to very good). The analysis of the 466 responses revealed that most developers consider steps to reproduce, stack traces, and test cases as helpful. The authors also show that bug reports containing stack traces get fixed sooner, and those easier to read have lower lifetimes.

The study of Soltani *et al.* (2020) aims to establish the significance of bug report elements. The authors interviewed 35 developers to gain insights into the importance of various contents in bug reports, followed by a survey applied to 305 developers. Based on the acquired data from these moments, the authors conclude that the essential elements are crash description, reproducing steps or test cases, and stack traces. They also evaluate the quality of bug reports of the 250 most popular projects on Github. Their analysis shows that crash reproducing steps, stack traces, fix suggestions, and user contents, have a statistically significant impact on bug resolution times between 76% to 33% of the projects.

Sasso *et al.* (2016) describes what makes a satisficing (a neologism combining the verbs to satisfy and to suffice) a bug report. The authors proposed a questionnaire to an open-source community. The authors gather the perception of how difficult it is to provide distinct kinds of information during the bug report record. They also mined content from Bugzilla and Jira to understand what users and developers collect and provide during the bug reporting. Based on more than 650,000 bug reports and the results from the questionnaire, the authors evaluate how the completeness of standard and project-specific attributes in a bug report related to its lifetime, similar to the BFT concept in our study. Finally, they highlight that number of words in the description and the summary are more correlated are the features that impact the prediction the most.

None of the presented studies evaluate the relationship between report links, priority and code-churn size, and the BFT. Also, none of them use Bayesian statistics or incorporate previous results from other studies into their analysis. For instance, the studies Zimmermann *et al.* (2010), Soltani *et al.* (2020), Sasso *et al.* (2016) apply a similar methodology when using surveys and questionnaires, and Zimmermann *et al.* (2010), Soltani *et al.* (2020) conclude similar things. This is a good example where studies using data on the same subject and trying to answer similar research questions can benefit from using previous results as priors for their study. If the results were modeled using Bayes statistics, providing a conclusion based on posterior distributions, one could continually use previous results to gather more evidence of previous findings. Also, none of the founded related work seems to present the impact of the analyzed features (i.e., the average impact in days of the existence or non-existence of an specific feature) numerically.

5 CONCLUSION AND FUTURE WORKS

This thesis presents three contributions regarding the bug reports process and fix-time estimation. The first is a public dataset composed from bug reports of 55 Apache Software Foundation projects. We describe the mining process and the tools we used. The dataset is presented in two perspectives: the static, composed by the last state snapshot of each report, and the dynamic, composed by every change and update that occurs on each report. We also discuss a characterizing analysis of some report characteristics. The presented data is used in the other two thesis contributions.

The second contribution is a study on how field updates impact the bug-fix time prediction using machine learning models. Based on the previously proposed dataset, we create a new dataset based on the final reports' state and their previous fields' changes and updates. We test several configurations to build different models: three machine learning algorithms (logistic regression, neural network, and Gaussian process), the use or not of data balance (use of original data, oversampling or undersampling), and different days thresholds to classify the bug reports as 1) more or less than five days to be fixed (two classes); 2) more or less than ten days to be fixed (two classes). The best f-measure values (we also present log-loss, accuracy, recall, and precision) are acquired using classification models, predicting the bug reports as more or less than five days to be fixed. Neural networks, linear regression, and Gaussian processes all present moderately similar results. However, Gaussian Processes outperforms the others in four projects (Hadoop MapReduce, Hadoop HDFS, Kafka, and Spark). For four other projects, linear regression presents the best results (Hadoop Core, Flink, Lucene, and Sol). The neural networks provide the best results for two projects, Hadoop Yarn and Zookeeper. Our approach uses the bug reports as patterns to train machine learning models, but with a particularity. The bug reports have changes and updates in their fields, from creation to resolution. We consider that for each field addition or update during its lifetime, we have a new report with more information and a shorter bug-fix time. This allows us to have more data and verify how the reports' updates impact the models' prediction capacity. Our experiments show that field updates impact the models' performance. We get the best results when predicting the resolution time at the initial report states (close to data creation), suitable for a practical scenario, and at the final report states. The results vary depending on the project. For the initial report's best estimations, we acquire f-measures between 0.63 up to 0.87, depending on the project. Our approach also outperforms the baseline work Zhang *et al.* (2013) using different sets of attributes and is also comparable to

similar works with other data. All selected attributes are easy to compute and understand, ideal for a real-world use scenario.

The third contribution are the results of a Bayesian Data Analysis on bug report data. We propose two models as the hypothesis to the data generation process and verify how 'links', 'priority' and 'code churn', all features present in the bug reports, are related to the BFT. Based on the obtained results, we gather evidence that priority plays no role in the bug fix time. In contrast, bug reports with higher values of code churn and bugs reports that relate to other bugs (with links) need at least double the time to be fixed compared to reports with smaller code churn values and those with no links.

We look to improve the three contributions in future works. Using the mining script, we can continually mine new bug reports from 2019 and posterior years to evaluate the approach proposed in the Chapter 3, also being able to evaluate new recorded bug reports in real time. We see opportunities to improve the bug fixing estimation approach results looking for means to define an impactful report update. As we consider every change and comment in the report as a new state, it can introduce several similar and noisy data, as a change of a specific field may be more impactful in the resolution time than others. The idea is to identify the updates that meaningfully modify the report information and content, demanding a new fix time estimation. Another oportunity is to estimate intervals of bug fixing time, not the classes. Some probabilistics models (as Gaussian process) provides a distribution of the value to be estimated as output. This can be used to suggest plausible intervals, not only point estimation. Finally, another idea to explore is the use of contextual information about others reports. The hyphoteses is to evaluate how a set of bug reports in the same window of time impacts one each other in therms of priorization and resolution.

For the BFT results, we intend to provide a hierarchical model at the project level. In this thesis, we fit two models (three for priority) for each project and feature using the 'specific model'. Using all project data in one single model, we can identify more reliable results. Also, we partialy explore the idea of alternative hierarchical models. In the future works, it is possible to provide new hierachical models and look for ways to select the ones that best fit the bug reporting process. Another ideia is propose a regression analysis and provide a more thoughtful view of all features, not only the selected in this thesis, and how they relate to the bug fix time.

BIBLIOGRAPHY

AKBARINASAJI, S.; CAGLAYAN, B.; BENER, A. Predicting bug-fixing time: A replication study using an open source software project. **Journal of Systems and Software**, v. 136, p. 173 – 186, 2018. ISSN 0164-1212. Disponível em: http://www.sciencedirect.com/science/article/pii/S0164121217300365. Acesso em: 12 de set. de 2019.

AL-ZUBAIDI, W. H. A.; DAM, H. K.; GHOSE, A.; LI, X. Multi-objective search-based approach to estimate issue resolution time. In: **Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2017. (PROMISE), p. 53–62. ISBN 9781450353052. Disponível em: https://doi.org/10.1145/3127005.3127011. Acesso em: 02 de ago. de 2019.

ALKHAZI, B.; DISTASI, A.; ALJEDAANI, W.; ALRUBAYE, H.; YE, X.; MKAOUER, M. W. Learning to rank developers for bug report assignment. **Applied Soft Computing**, v. 95, p. 106667, 2020. ISSN 1568-4946. Disponível em: https://www.sciencedirect.com/science/article/pii/S1568494620306050. Acesso em: 10 de dez. de 2020.

ALMHANA, R.; FERREIRA, T.; KESSENTINI, M.; SHARMA, T. Understanding and characterizing changes in bugs priority: The practitioners' perceptive. In: **2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [*S. l.*: *s. n.*], 2020. p. 87–97.

ARDIMENTO, P.; BILANCIA, M.; MONOPOLI, S. Predicting bug-fix time: Using standard versus topic-based text categorization techniques. In: CALDERS, T.; CECI, M.; MALERBA, D. (Ed.). **Discovery Science**. Cham: Springer International Publishing, 2016. p. 167–182. ISBN 978-3-319-46307-0.

ASSAR, S.; BORG, M.; PFAHL, D. Using text clustering to predict defect resolution time: A conceptual replication and an evaluation of prediction accuracy. **Empirical Softw. Engg.**, Kluwer Academic Publishers, USA, v. 21, n. 4, p. 1437–1475, ago. 2016. ISSN 1382-3256. Disponível em: https://doi.org/10.1007/s10664-015-9391-7. Acesso em: 03 de mar. de 2020.

BAYSAL, O.; HOLMES, R.; GODFREY, M. W. Situational awareness: Personalizing issue tracking systems. In: **Proceedings of the 2013 International Conference on Software Engineering**. [*S. l.*]: IEEE Press, 2013. (ICSE '13), p. 1185–1188. ISBN 9781467330763.

BHATTACHARYA, P.; NEAMTIU, I. Bug-fix time prediction models: Can we do better? In: **Proceedings of the 8th Working Conference on Mining Software Repositories**. New York, NY, USA: Association for Computing Machinery, 2011. (MSR '11), p. 207–210. ISBN 9781450305747. Disponível em: https://doi.org/10.1145/1985441.1985472. Acesso em: 01 de out. de 2019.

BORG, M.; PFAHL, D.; RUNESON, P. Analyzing networks of issue reports. In: **2013 17th European Conference on Software Maintenance and Reengineering**. [*S. l.*: *s. n.*], 2013. p. 79–88.

BRADY, F. **Cambridge University report on cost of software faults, Press release, 2013**. 2013. Disponível em: http://www.prweb.com/releases/2013/1/prweb10298185.htm. Acesso em: 02 de jan. de 2020.

CANFORA, G.; CECCARELLI, M.; CERULO, L.; PENTA, M. D. How long does a bug survive? an empirical study. In: **2011 18th Working Conference on Reverse Engineering**. [*S. l.*: *s. n.*], 2011. p. 191–200. ISSN 2375-5369.

CATOLINO, G.; PALOMBA, F.; ZAIDMAN, A.; FERRUCCI, F. Not all bugs are the same: Understanding, characterizing, and classifying bug types. **Journal of Systems and Software**, v. 152, p. 165–181, 2019. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121219300536. Acesso em: 06 de jul. de 2019.

CHAWLA, N. V.; BOWYER, K. W.; HALL, L. O.; KEGELMEYER, W. P. Smote: synthetic minority over-sampling technique. **Journal of artificial intelligence research**, v. 16, p. 321–357, 2002.

EBRAHIMI, N.; TRABELSI, A.; ISLAM, M. S.; HAMOU-LHADJ, A.; KHANMOHAMMADI, K. An hmm-based approach for automatic detection and classification of duplicate bug reports. **Information and Software Technology**, v. 113, p. 98 – 109, 2019. ISSN 0950-5849. Disponível em: http://www.sciencedirect.com/science/article/pii/S095058491930117X. Acesso em: 24 de abr. de 2020.

FURIA, C. A.; FELDT, R.; TORKAR, R. Bayesian data analysis in empirical software engineering research. **IEEE Transactions on Software Engineering**, v. 47, n. 9, p. 1786–1810, 2021.

GELMAN, A.; CARLIN, J.; STERN, H.; DUNSON, D.; VEHTARI, A.; RUBIN, D. **Bayesian Data Analysis, Third Edition**. Taylor & Francis, 2013. (Chapman & Hall/CRC Texts in Statistical Science). ISBN 9781439840955. Disponível em: https://books.google.com.br/books?id=ZXL6AQAAQBAJ. Acesso em: 17 de set. de 2021.

GELMAN, A.; VEHTARI, A.; SIMPSON, D.; MARGOSSIAN, C. C.; CARPENTER, B.; YAO, Y.; KENNEDY, L.; GABRY, J.; BüRKNER, P.-C.; MODRáK, M. **Bayesian Workflow**. 2020.

GOUES, C. L.; PRADEL, M.; ROYCHOUDHURY, A.; CHANDRA, S. Automatic program repair. **IEEE Software**, v. 38, n. 4, p. 22–27, 2021.

GUO, P. J.; ZIMMERMANN, T.; NAGAPPAN, N.; MURPHY, B. “not my bug!” and other reasons for software bug report reassignments. In: **Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work**. New York, NY, USA: Association for Computing Machinery, 2011. (CSCW ’11), p. 395–404. ISBN 9781450305563. Disponível em: https://doi.org/10.1145/1958824.1958887. Acesso em: 02 de fev. de 2019.

HABAYEB, M.; MIRANSKYY, A.; MURTAZA, S. S.; BUCHANAN, L.; BENER, A. The firefox temporal defect dataset. In: **2015 IEEE/ACM 12th Working Conference on Mining Software Repositories**. [*S. l.*: *s. n.*], 2015. p. 498–501. ISSN 2160-1852.

HABAYEB, M.; MURTAZA, S. S.; MIRANSKYY, A.; BENER, A. B. On the use of hidden markov model to predict the time to fix bugs. **IEEE Transactions on Software Engineering**, v. 44, n. 12, p. 1224–1244, Dec 2018. ISSN 2326-3881.

HAMILL, M.; GOSEVA-POPSTOJANOVA, K. Analyzing and predicting effort associated with finding and fixing software faults. **Information and Software Technology**, v. 87, p. 1 – 18, 2017. ISSN 0950-5849. Disponível em: http://www.sciencedirect.com/science/article/pii/S0950584917300290. Acesso em: 27 de nov. de 2020.

HAUGE, O.; AYALA, C.; CONRADI, R. Adoption of open source software in software-intensive organizations - a systematic literature review. **Inf. Softw. Technol.**, Butterworth-Heinemann, USA, v. 52, n. 11, p. 1133–1154, nov. 2010. ISSN 0950-5849. Disponível em: https://doi.org/10.1016/j.infsof.2010.05.008. Acesso em: 5 de jan. de 2019.

HEARD, N. A.; RUBIN-DELANCHY, P. Choosing between methods of combining
p
-values. **Biometrika**, v. 105, n. 1, p. 239–246, 01 2018. ISSN 0006-3444. Disponível em: https://doi.org/10.1093/biomet/asx076. Acesso em: 10 de ago. de 2021.

HENSMAN, J.; FUSI, N.; LAWRENCE, N. D. Gaussian processes for big data. In: **Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence**. [*S. l.*: *s. n.*], 2013. p. 282–290.

HERZIG, K.; JUST, S.; ZELLER, A. It's not a bug, it's a feature: How misclassification impacts bug prediction. In: **Proceedings of the 2013 International Conference on Software Engineering**. [*S. l.*]: IEEE Press, 2013. (ICSE '13), p. 392–401. ISBN 9781467330763.

HOOIMEIJER, P.; WEIMER, W. Modeling bug report quality. In: **Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2007. (ASE '07), p. 34–43. ISBN 9781595938824. Disponível em: https://doi.org/10.1145/1321631.1321639. Acesso em: 19 de dez. de 2020.

HU, H.; ZHANG, H.; XUAN, J.; SUN, W. Effective bug triage based on historical bug-fix information. In: **2014 IEEE 25th International Symposium on Software Reliability Engineering**. [*S. l.*: *s. n.*], 2014. p. 122–132. ISSN 2332-6549.

KARIM, M. R.; IHARA, A.; YANG, X.; IIDA, H.; MATSUMOTO, K. Understanding key features of high-impact bug reports. In: **2017 8th International Workshop on Empirical Software Engineering in Practice (IWESEP)**. [*S. l.*: *s. n.*], 2017. p. 53–58.

KIM, S.; WHITEHEAD, E. J. How long did it take to fix bugs? In: **Proceedings of the 2006 International Workshop on Mining Software Repositories**. New York, NY, USA: Association for Computing Machinery, 2006. (MSR '06), p. 173–174. ISBN 1595933972. Disponível em: https://doi.org/10.1145/1137983.1138027. Acesso em: 21 de out. de 2020.

KRUSCHKE, J. **Doing Bayesian Data Analysis (Second Edition)**. Boston: Academic Press, 2015.

LAMKANFI, A.; PÉREZ, J.; DEMEYER, S. The eclipse and mozilla defect tracking dataset: A genuine dataset for mining bug information. In: **2013 10th Working Conference on Mining Software Repositories (MSR)**. [*S. l.*: *s. n.*], 2013. p. 203–206. ISSN 2160-1860.

LAZAR, A.; RITCHEY, S.; SHARIF, B. Improving the accuracy of duplicate bug report detection using textual similarity measures. In: **Proceedings of the 11th Working Conference on Mining Software Repositories**. New York, NY, USA: Association for Computing Machinery, 2014. (MSR 2014), p. 308–311. ISBN 9781450328630. Disponível em: https://doi.org/10.1145/2597073.2597088. Acesso em: 12 de out. de 2019.

LENARDUZZI, V.; Taibi, D.; Tosi, D.; Lavazza, L.; Morasca, S. Open source software evaluation, selection, and adoption: a systematic literature review. In: **2020 46th Euromicro**

Conference on Software Engineering and Advanced Applications (SEAA). [*S. l.*: *s. n.*], 2020. p. 437–444.

MCELREATH, R. **Statistical Rethinking: A Bayesian Course with Examples in R and Stan**. 2nd. ed. [*S. l.*]: Chapman and Hall/CRC, 2020.

ORTU, M.; DESTEFANIS, G.; ADAMS, B.; MURGIA, A.; MARCHESI, M.; TONELLI, R. The jira repository dataset: Understanding social aspects of software development. In: **Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering**. New York, NY, USA: ACM, 2015. (PROMISE '15), p. 1:1–1:4. ISBN 978-1-4503-3715-1. Disponível em: http://doi.acm.org/10.1145/2810146.2810147. Acesso em: 03 de jan. de 2019.

RAHMAN, S.; GANGULY, K. K.; SAKIB, K. An improved bug localization using structured information retrieval and version history. In: **2015 18th International Conference on Computer and Information Technology (ICCIT)**. [*S. l.*: *s. n.*], 2015. p. 190–195.

RAJA, U. All complaints are not created equal: text analysis of open source software defect reports. **Empirical Software Engineering**, v. 18, n. 1, p. 117–138, Feb 2013. ISSN 1573-7616. Disponível em: https://doi.org/10.1007/s10664-012-9197-9. Acesso em: 05 de out. de 2019.

SAHA, R. K.; KHURSHID, S.; PERRY, D. E. An empirical study of long lived bugs. In: **2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)**. [*S. l.*: *s. n.*], 2014. p. 144–153.

SAHA, R. K.; KHURSHID, S.; PERRY, D. E. Understanding the triaging and fixing processes of long lived bugs. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 65, n. C, p. 114–128, set. 2015. ISSN 0950-5849. Disponível em: http://dx.doi.org/10.1016/j.infsof.2015.03.002. Acesso em: 06 de nov. de 2019.

SASSO, T. D.; MOCCI, A.; LANZA, M. What makes a satisficing bug report? In: **2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)**. [*S. l.*: *s. n.*], 2016. p. 164–174.

SERRANO, N.; CIORDIA, I. Bugzilla, itracker, and other bug trackers. **IEEE Software**, v. 22, n. 2, p. 11–13, 2005.

SHARMA, M.; BEDI, P.; CHATURVEDI, K. K.; SINGH, V. B. Predicting the priority of a reported bug using machine learning techniques and cross project validation. In: **2012 12th International Conference on Intelligent Systems Design and Applications (ISDA)**. [*S. l.*: *s. n.*], 2012. p. 539–545. ISSN 2164-7143. Acesso em: 17 de set. de 2019.

SHARMA, M.; KUMARI, M.; SINGH, V. B. Multi-attribute dependent bug severity and fix time prediction modeling. **International Journal of System Assurance Engineering and Management**, v. 10, n. 5, p. 1328–1352, Oct 2019. ISSN 0976-4348. Disponível em: https://doi.org/10.1007/s13198-019-00888-5. Acesso em: 17 de nov. de 2019.

SHOKRIPOUR, R.; ANVIK, J.; KASIRUN, Z. M.; ZAMANI, S. A time-based approach to automatic bug report assignment. **J. Syst. Softw.**, Elsevier Science Inc., USA, v. 102, n. C, p. 109–122, abr. 2015. ISSN 0164-1212. Disponível em: https://doi.org/10.1016/j.jss.2014.12.049. Acesso em: 06 de fev. de 2020.

SOLTANI, M.; HERMANS, F.; BÄCK, T. The significance of bug report elements. **Empirical Software Engineering**, v. 25, n. 6, p. 5255–5294, Nov 2020. ISSN 1573-7616. Disponível em: https://doi.org/10.1007/s10664-020-09882-z. Acesso em: 09 de out. de 2021.

SPADINI, D.; ANICHE, M.; BACCHELLI, A. PyDriller: Python framework for mining software repositories. In: **Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018**. New York, New York, USA: ACM Press, 2018. p. 908–911. ISBN 9781450355735. Disponível em: http://dl.acm.org/citation.cfm?doid=3236024.3264598. Acesso em: 07 de fev. de 2019.

THOMPSON, C. A.; MURPHY, G. C.; PALYART, M.; GAšPARIC, M. How software developers use work breakdown relationships in issue repositories. In: **2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)**. [*S. l.*: *s. n.*], 2016. p. 281–285.

THUNG, F. Automatic prediction of bug fixing effort measured by code churn size. In: **Proceedings of the 5th International Workshop on Software Mining**. New York, NY, USA: Association for Computing Machinery, 2016. (SoftwareMining 2016), p. 18–23. ISBN 9781450345118. Disponível em: https://doi.org/10.1145/2975961.2975964. Acesso em: 23 de jan. de 2020.

TIAN, Y.; LO, D.; SUN, C. Drone: Predicting priority of reported bugs by multi-factor analysis. In: **2013 IEEE International Conference on Software Maintenance**. [*S. l.*: *s. n.*], 2013. p. 200–209. ISSN 1063-6773.

TIAN, Y.; LO, D.; XIA, X.; SUN, C. Automated prediction of bug report priority using multi-factor analysis. **Empirical Softw. Engg.**, Kluwer Academic Publishers, USA, v. 20, n. 5, p. 1354–1383, out. 2015. ISSN 1382-3256. Disponível em: https://doi.org/10.1007/s10664-014-9331-y. Acesso em: 29 de nov. de 2020.

TOMOVA, M. T.; RATH, M.; MäDER, P. Poster: Use of trace link types in issue tracking systems. In: **2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)**. [*S. l.*: *s. n.*], 2018. p. 181–182.

TORKAR, R.; FURIA, C. A.; FELDT, R.; NETO, F. Gomes de O.; GREN, L.; LENBERG, P.; ERNST, N. A. A method to assess and argue for practical significance in software engineering. **IEEE Transactions on Software Engineering**, p. 1–1, 2021.

UMER, Q.; LIU, H.; ILLAHI, I. Cnn-based automatic prioritization of bug reports. **IEEE Transactions on Reliability**, v. 69, n. 4, p. 1341–1354, 2020.

UMER, Q.; LIU, H.; SULTAN, Y. Emotion based automated priority prediction for bug reports. **IEEE Access**, v. 6, p. 35743–35752, 2018. ISSN 2169-3536.

VIEIRA, R.; SILVA, A. da; ROCHA, L.; GOMES, J. a. P. From reports to bug-fix commits: A 10 years dataset of bug-fixing activity from 55 apache's open source projects. In: **Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering**. New York, NY, USA: ACM, 2019. (PROMISE'19), p. 80–89. ISBN 978-1-4503-7233-6. Disponível em: http://doi.acm.org/10.1145/3345629.3345639. Acesso em: 17 de nov. de 2019.

WEISS, C.; PREMRAJ, R.; ZIMMERMANN, T.; ZELLER, A. How long will it take to fix this bug? In: **Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)**. [*S. l.*: *s. n.*], 2007. p. 1–1. ISSN 2160-1860.

WOHLIN, C.; RUNESON, P.; HST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLN, A. **Experimentation in Software Engineering**. [*S. l.*]: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434.

XU, Y.; ZHOU, M. A multi-level dataset of linux kernel patchwork. In: **2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)**. [*S. l.*: *s. n.*], 2018. p. 54–57. ISSN 2574-3864.

ZHANG, F.; KHOMH, F.; ZOU, Y.; HASSAN, A. E. An empirical study on factors impacting bug fixing time. In: **2012 19th Working Conference on Reverse Engineering**. [*S. l.*: *s. n.*], 2012. p. 225–234. ISSN 2375-5369.

ZHANG, H.; GONG, L.; VERSTEEG, S. Predicting bug-fixing time: An empirical study of commercial software projects. In: **2013 35th International Conference on Software Engineering (ICSE)**. [*S. l.*: *s. n.*], 2013. p. 1042–1051. ISSN 1558-1225.

ZHANG, X.; CHEN, X.; YAO, L.; GE, C.; DONG, M. Deep neural network hyperparameter optimization with orthogonal array tuning. In: GEDEON, T.; WONG, K. W.; LEE, M. (Ed.). **Neural Information Processing**. Cham: Springer International Publishing, 2019. p. 287–295. ISBN 978-3-030-36808-1.

ZHANG, X.; YAO, L.; HUANG, C.; SHENG, Q. Z.; WANG, X. Intent recognition in smart living through deep recurrent neural networks. In: LIU, D.; XIE, S.; LI, Y.; ZHAO, D.; EL-ALFY, E.-S. M. (Ed.). **Neural Information Processing**. Cham: Springer International Publishing, 2017. p. 748–758. ISBN 978-3-319-70096-0.

ZHOU, J.; CUI, G.; ZHANG, Z.; YANG, C.; LIU, Z.; SUN, M. Graph neural networks: A review of methods and applications. **CoRR**, abs/1812.08434, 2018. Disponível em: http://arxiv.org/abs/1812.08434. Acesso em: 01 de out. de 2021.

ZHU, J.; ZHOU, M.; MEI, H. Multi-extract and multi-level dataset of mozilla issue tracking history. In: **2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)**. [*S. l.*: *s. n.*], 2016. p. 472–475.

ZIMMERMANN, T.; PREMRAJ, R.; BETTENBURG, N.; JUST, S.; SCHROTER, A.; WEISS, C. What makes a good bug report? **IEEE Transactions on Software Engineering**, v. 36, n. 5, p. 618–643, Sep. 2010. ISSN 2326-3881.

APPENDIX A – DATASET FEATURES TABLE

Static Perspective

We present the dataset attributes in this appendix. Tables 23 and 24 shows the static dataset features, as discussed in Chapter 2.

Table 23 – Static dataset fields - Jira.

From	Type	Field
Jira (30)	General (10)	Project
		Owner
		Manager
		Category
		Key
		Priority
		Status
		Reporter
		Assignee
		Components
	Link (2)	InwardIssueLinks
		OutwardIssueLinks
	Summation (4)	NoComments
		NoWatchers
		NoAttachments
		NoAttachedPatches
	Text (3)	SummaryTopWords
		DescriptionTopWords
		CommentsTopWords
	Time (8)	CreationDate
		ResolutionDate
		FirstCommentDate
		LastCommentDate
		FirstAttachmentDate
		LastAttachmentDate
		FirstAttachedPatchDate
		LastAttachedPatchDate
	Versioning (2)	AffectsVersions
		FixVersions

Table 24 – Static dataset fields - Git

From	Type	Field
Git (24)	Text (1)	CommitsMessagesTopWords
	Versioning (1)	HasMergeCommit
	Summation (3)	NoCommits
		NoAuthors
		NoCommitters
	Time (4)	AuthorsFirstCommitDate
		AuthorsLastCommitDate
		CommittersFirstCommitDate
		CommittersLastCommitDate
	Source (15)	NonSrcAddFiles
		NonSrcDelFiles
		NonSrcModFiles
		NonSrcAddLines
		NonSrcDelLines
		SrcAddFiles
		SrcDelFiles
		SrcModFiles
		SrcAddLines
		SrcDelLines
		TestAddFiles
		TestDelFiles
		TestModFiles
		TestAddLines
		TestDelLines

Dynamic Perspective

The Tables 25, 26, and 27 show the features in the dynamic files of the dataset.

Table 25 – Changelog dataset fields

Field	From	Type
Jira (9)	General (6)	Project
		Manager
		Category
		Key
		Author
		Field
	Time (1)	ChangeDate
	Text (2)	From
		To

Table 26 – Comment-log dataset fields

Field	From	Type
Jira (7)	General (5)	`Project`
		`Manager`
		`Category`
		`Key`
		`Author`
	Time (1)	`CommentDate`
	Text (1)	`Content`

Table 27 – Commit-log dataset fields

Field	From	Type
Jira (4)	General (4)	`Project`
		`Manager`
		`Category`
		`Key`
Git (18)	Versioning (2)	`CommitHash`
		`IsMergeCommit`
	General (2)	`Author`
		`Committer`
	Time (2)	`AuthorDate`
		`CommitterDate`
	Text (1)	`CommitMessageTopWords`
	Source (11)	`FileName`
		`FilePath`
		`ChangeType`
		`IsSrcFile`
		`IsTestFile`
		`AddLines`
		`DelLines`
		`NoMethods`
		`LoC`
		`CyC`
		`NoTokens`

APPENDIX B – COMPLETE 'SPECIFIC-MODEL' RESULTS

This appendix presents the complete results obtained using the 'specific-model' in each project independently. Fig. 25 and Tables 28 and 29 present the μ posterior distributions and their summarizations, for 'links' results.

The first thing to notice is no standard behavior across all projects. However, in most projects, there is a trend that the reports with links have a higher resolution time value. There are extreme cases where the oppositive occurs, as in Madlib and Oozie, and cases where the difference between both groups is not as evident as Systemml, Helix, and Lang. For most projects, the uncertainty about the true value of μ is higher in reports with links compared to the values of reports with no links. This can be justified as all projects, except Derby, have more reports with no links than reports with links (see Fig. 6).

Fig. 26 and Tables 30, 31, and 32 present the μ posterior distributions and their summarizations, for 'priority' results.

Similar to the links, there is no standard behavior across all projects. The only consistent thing between all the projects is that the variance of μ for medium priority is smaller than the other priorities group. This can be justified as most reports have a medium priority. Once again, the results give us a general picture of each project's behavior but do not help us to verify a bug reports global behavior.

Finally, Fig. 27 and Tables 33, and 34 present the μ posterior distributions and their summarizations, for 'code-churn' results.

The observations are very similar to the results from 'links'. In most projects, there is a trend that the reports with the above code churn median have a higher resolution time value than those with the below code churn median value. There are extreme cases where the oppositive occurs, as in SSHD and Maven, and cases where the difference between both groups is not as evident as Commons IO, FORTRESS, and Commons Collections. Once again, the results give us a general picture of each project's behavior but do not help us to verify a bug reports global behavior.

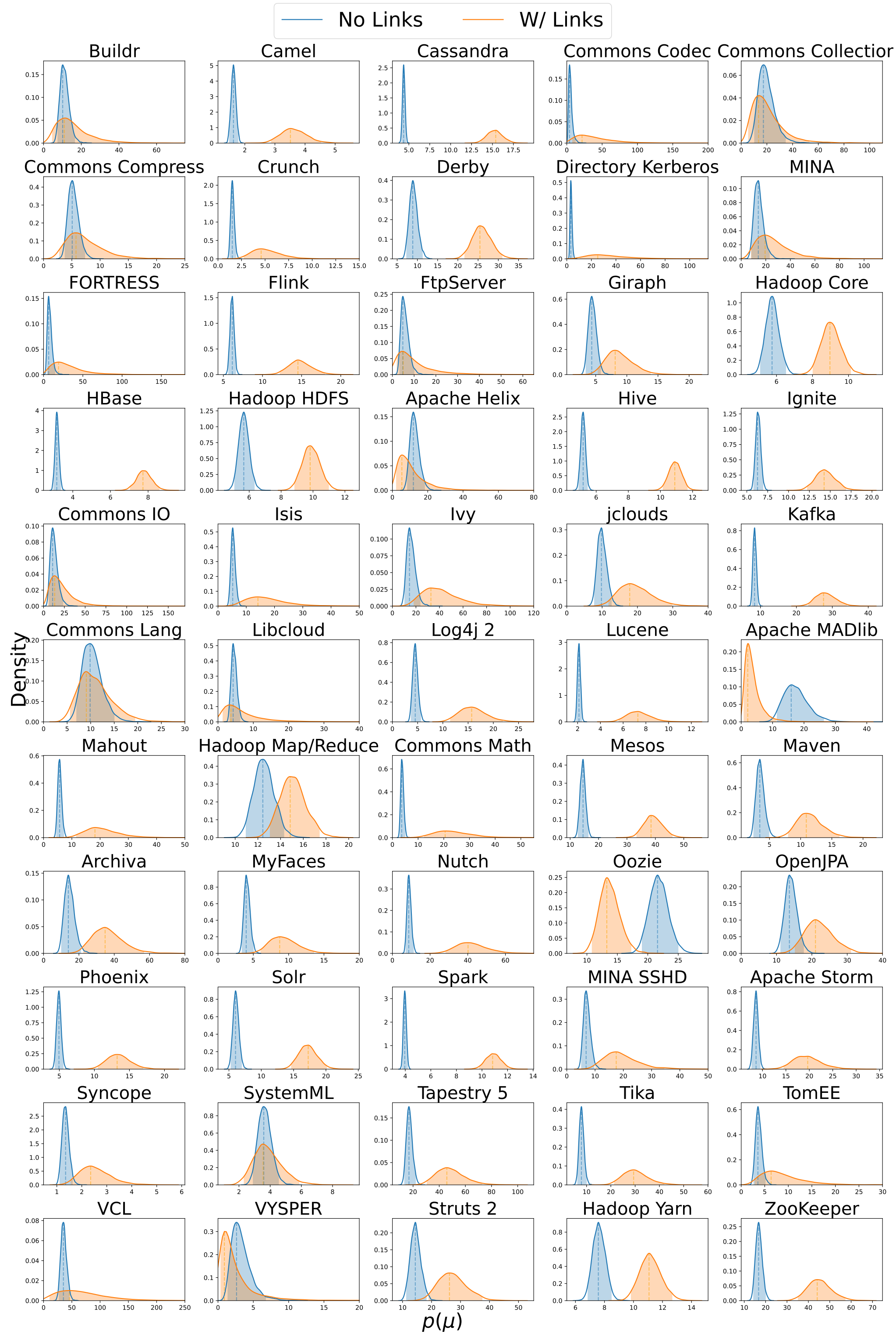


Figure 25 – μ posterior distributions - ‘specific-models’, ‘links’ results, for all 55 projects. The average bug fixing time of reports with no links vs. those with links. The results diverge depending on the selected project.

Table 28 – μ posterior distribution summary, 'specific-models', 'No Links' results (55 projects).

No Links (a)					
Project	CI_L	CI_U	μ_{MAP}	E[F1(a,b)]	E[F2(a,b)]
Buildr	7.29	16.72	10.15	-4.50	0.36
Camel	1.47	1.79	1.62	-2.00	0.00
Cassandra	4.10	4.70	4.38	-10.88	0.00
Commons Codec	1.72	12.63	3.92	-34.23	0.05
Commons Collections	9.89	34.91	17.29	-0.91	0.53
Commons Compress	3.70	7.37	5.07	-2.01	0.30
Crunch	1.21	1.95	1.52	-3.65	0.00
Derby	7.24	11.35	8.86	-16.93	0.00
Directory Kerberos	2.21	5.32	3.29	-31.37	0.00
MINA	8.16	23.39	13.92	-12.19	0.20
FORTRESS	3.60	14.98	6.50	-22.71	0.09
Flink	5.62	6.62	6.14	-8.72	0.00
FtpServer	3.11	10.21	4.80	-4.47	0.38
Giraph	3.36	5.84	4.36	-4.40	0.01
Hadoop Core	5.09	6.54	5.75	-3.26	0.00
HBase	2.96	3.38	3.15	-4.63	0.00
Hadoop HDFS	5.06	6.35	5.66	-4.21	0.00
Apache Helix	8.28	18.42	11.88	1.85	0.68
Hive	4.90	5.47	5.19	-5.73	0.00
Ignite	5.78	6.98	6.28	-8.07	0.00
Commons IO	6.05	23.66	10.89	-8.82	0.29
Isis	4.01	7.08	5.21	-12.11	0.01
Ivy	10.10	24.00	14.51	-25.72	0.03
jclouds	7.69	12.75	9.80	-9.44	0.01
Kafka	7.55	9.36	8.37	-19.82	0.00
Commons Lang	6.87	15.10	9.88	-0.68	0.46
Libcloud	3.23	6.34	4.29	-2.18	0.43
Log4j 2	3.62	5.68	4.53	-11.41	0.00
Lucene	1.87	2.38	2.12	-5.27	0.00
Apache MADlib	11.18	26.38	15.92	13.69	0.99
Mahout	4.45	7.24	5.68	-15.01	0.00
Hadoop Map/Reduce	10.92	14.35	12.44	-2.55	0.04
Commons Math	2.82	5.09	3.62	-19.88	0.00
Mesos	12.71	16.47	14.54	-24.77	0.00
Maven	2.40	4.99	3.45	-7.98	0.00
Archiva	9.93	21.24	14.02	-21.71	0.00
MyFaces	3.37	5.05	3.99	-5.23	0.00
Nutch	6.96	11.35	8.69	-33.15	0.00
Oozie	18.97	25.04	21.60	8.07	1.00
OpenJPA	11.05	17.77	13.64	-8.06	0.03
Phoenix	4.34	5.59	4.95	-8.53	0.00
Solr	5.25	7.02	6.02	-11.12	0.00
Spark	3.75	4.20	3.96	-6.92	0.00
MINA SSHD	5.05	9.82	6.84	-12.22	0.01
Apache Storm	7.64	9.60	8.63	-10.95	0.00
Syncope	1.10	1.64	1.35	-1.19	0.01
SystemML	2.88	4.56	3.59	-0.20	0.45
Tapestry 5	13.19	22.24	16.88	-31.33	0.00
Tika	6.17	10.08	7.88	-22.20	0.00
TomEE	2.58	5.20	3.55	-5.00	0.10
VCL	26.69	46.43	35.09	-33.03	0.24
VYSPER	1.35	6.81	2.63	0.81	0.72
Struts 2	11.25	18.35	14.40	-12.74	0.00
Hadoop Yarn	6.83	8.50	7.58	-3.54	0.00
ZooKeeper	14.16	20.10	16.72	-28.41	0.00

Table 29 – μ posterior distribution summary, 'specific-models', 'With Links' results (55 projects).

	With Links (b)				
Project	CI_L	CI_U	μ_{MAP}	E[F1(b,a)]	E[F2(b,a)]
Buildr	3.82	40.61	11.01	4.50	0.64
Camel	2.86	4.54	3.51	2.00	1.00
Cassandra	13.54	17.17	15.39	10.88	1.00
Commons Codec	5.70	119.45	21.60	34.23	0.95
Commons Collections	5.11	54.14	13.54	0.91	0.47
Commons Compress	2.69	15.41	5.74	2.01	0.70
Crunch	2.71	8.97	4.58	3.65	1.00
Derby	21.55	31.08	25.46	16.93	1.00
Directory Kerberos	10.00	86.07	25.18	31.37	0.99
MINA	7.61	63.74	19.43	12.19	0.80
FORTRESS	4.99	87.60	19.11	22.71	0.91
Flink	12.17	17.99	14.53	8.72	1.00
FtpServer	1.01	37.36	4.94	4.47	0.62
Giraph	5.24	13.83	8.13	4.40	0.99
Hadoop Core	8.01	10.16	8.97	3.26	1.00
HBase	7.02	8.61	7.73	4.63	1.00
Hadoop HDFS	8.83	11.00	9.82	4.21	1.00
Apache Helix	1.61	33.38	5.37	-1.85	0.32
Hive	10.13	11.70	10.89	5.73	1.00
Ignite	12.20	16.94	14.26	8.07	1.00
Commons IO	4.70	59.68	13.39	8.82	0.70
Isis	7.03	37.14	14.14	12.11	0.99
Ivy	17.72	82.94	32.73	25.72	0.97
jclouds	11.75	29.91	17.81	9.44	0.99
Kafka	23.21	34.11	27.57	19.82	1.00
Commons Lang	5.52	19.45	9.08	0.68	0.54
Libcloud	0.79	23.28	3.69	2.18	0.56
Log4j 2	11.34	21.99	15.71	11.41	1.00
Lucene	5.56	9.69	7.30	5.27	1.00
Apache MADlib	0.71	11.25	2.08	-13.69	0.01
Mahout	11.39	34.27	18.18	15.01	1.00
Hadoop Map/Reduce	13.01	17.45	14.85	2.55	0.96
Commons Math	12.10	41.51	20.54	19.88	1.00
Mesos	33.29	46.10	38.47	24.77	1.00
Maven	7.90	16.25	10.90	7.98	1.00
Archiva	22.05	57.00	34.84	21.71	1.00
MyFaces	5.98	13.69	8.73	5.23	1.00
Nutch	27.95	60.20	40.16	33.15	1.00
Oozie	10.79	17.29	13.27	-8.07	0.00
OpenJPA	15.06	31.22	21.02	8.06	0.97
Phoenix	10.47	17.04	13.25	8.53	1.00
Solr	14.61	20.14	17.26	11.12	1.00
Spark	9.76	12.06	10.83	6.92	1.00
MINA SSHD	10.03	34.61	17.49	12.22	0.99
Apache Storm	14.39	26.10	19.65	10.95	1.00
Syncope	1.54	3.98	2.36	1.19	0.99
SystemML	2.30	5.88	3.55	0.20	0.55
Tapestry 5	30.43	72.88	45.84	31.33	1.00
Tika	21.22	42.10	29.38	22.20	1.00
TomEE	2.81	20.24	6.35	5.00	0.90
VCL	8.11	185.85	45.60	33.03	0.75
VYSPER	0.21	11.32	0.90	-0.81	0.28
Struts 2	18.81	37.55	26.18	12.74	1.00
Hadoop Yarn	9.81	12.71	11.07	3.54	1.00
ZooKeeper	35.01	57.39	44.14	28.41	1.00

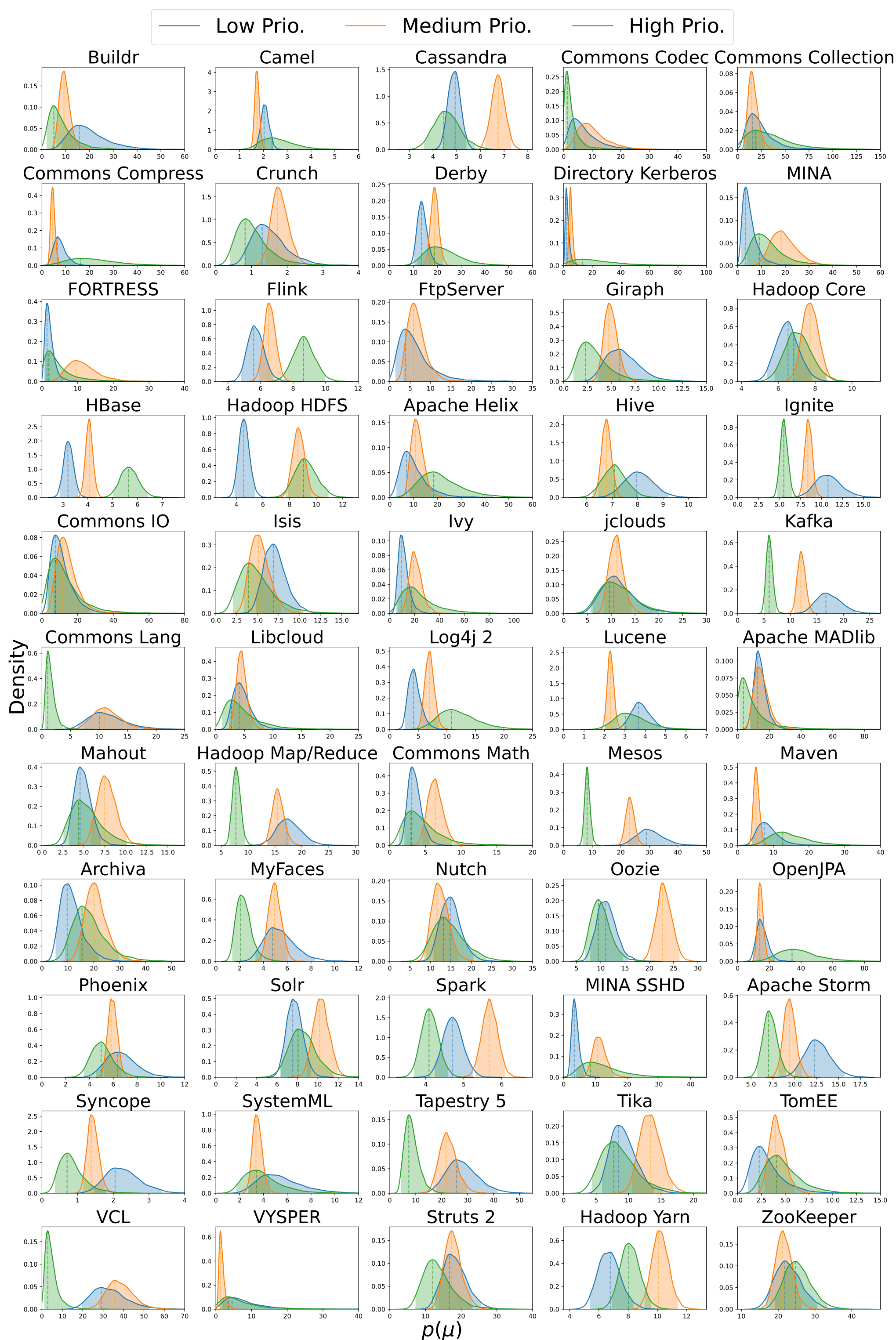


Figure 26 – μ posterior distributions - ‘specific-models’, ‘priority’ results, for all 55 projects. The average BFT of three priority levels: low, medium, and high. The results diverge depending on the selected project.

Table 30 – μ posterior distribution summary, ‘specific-models’, ‘Low Priority’ results (55 projects).

Low Priority (a)							
Project	CI_L	CI_U	μ_{MAP}	E[F1(a,b)]	E[F1(a,c)]	E[F2(a,b)]	E[F2(a,c)]
Buildr	8.25	39.87	15.73	10.09	11.41	0.91	0.89
Camel	1.76	2.42	2.03	0.34	-0.56	0.96	0.23
Cassandra	4.41	5.42	4.93	-1.85	0.30	0.00	0.70
Commons Codec	1.28	23.22	3.68	-2.96	4.74	0.32	0.82
Commons Collections	6.05	63.38	15.95	7.17	-11.07	0.65	0.36
Commons Compress	3.57	14.42	6.58	3.12	-14.18	0.87	0.08
Crunch	0.79	2.70	1.30	-0.28	0.46	0.27	0.77
Derby	10.12	18.28	13.28	-5.22	-8.86	0.03	0.11
Directory Kerberos	0.74	6.87	1.90	-2.40	-23.09	0.11	0.05
MINA	1.36	14.05	3.39	-14.55	-7.77	0.01	0.13
FORTRESS	0.57	6.05	1.42	-9.20	-2.70	0.01	0.32
Flink	4.79	6.79	5.58	-0.86	-2.97	0.09	0.00
FtpServer	1.37	18.21	3.82	-0.28	-	0.40	-
Giraph	3.41	10.38	5.89	1.27	2.95	0.73	0.89
Hadoop Core	5.37	7.82	6.53	-1.25	-0.59	0.06	0.27
HBase	2.85	3.61	3.20	-0.84	-2.46	0.00	0.00
Hadoop HDFS	3.85	5.42	4.55	-4.11	-4.65	0.00	0.00
Apache Helix	2.68	24.18	7.09	-1.66	-11.66	0.33	0.12
Hive	7.00	9.22	7.96	1.28	0.96	0.99	0.90
Ignite	8.06	14.25	10.76	2.39	5.31	0.94	1.00
Commons IO	3.38	26.97	7.48	-3.04	-1.83	0.33	0.47
Isis	4.77	10.13	6.84	1.80	2.12	0.84	0.82
Ivy	5.05	21.63	9.20	-9.91	-12.76	0.06	0.19
jclouds	6.05	20.13	10.55	0.45	-0.08	0.51	0.50
Kafka	13.31	22.37	16.70	5.24	11.27	0.99	1.00
Commons Lang	6.31	19.23	10.08	0.08	9.94	0.49	1.00
Libcloud	2.26	8.58	4.09	0.10	-0.16	0.49	0.56
Log4j 2	2.63	6.96	4.14	-2.61	-7.70	0.04	0.00
Lucene	2.95	4.80	3.66	1.49	0.45	1.00	0.71
Apache MADlib	7.94	23.34	12.66	-0.84	4.65	0.45	0.77
Mahout	3.21	7.05	4.54	-2.86	-0.53	0.03	0.44
Hadoop Map/Reduce	13.46	22.53	17.22	2.09	9.80	0.79	1.00
Commons Math	2.03	5.61	3.08	-2.90	-1.07	0.02	0.40
Mesos	22.37	39.93	28.94	6.98	21.94	0.94	1.00
Maven	3.99	16.05	7.47	3.26	-6.12	0.87	0.15
Archiva	5.08	22.08	9.72	-9.31	-6.86	0.06	0.18
MyFaces	3.31	8.36	4.77	0.43	3.03	0.59	0.99
Nutch	10.66	20.53	14.84	2.55	0.34	0.78	0.55
Oozie	7.78	15.64	11.01	-11.84	1.18	0.00	0.67
OpenJPA	9.58	23.53	13.97	1.04	-23.09	0.58	0.02
Phoenix	4.51	9.47	6.36	0.73	1.59	0.69	0.84
Solr	6.34	9.36	7.54	-2.61	-0.82	0.01	0.31
Spark	4.23	5.24	4.70	-0.97	0.61	0.00	0.96
MINA SSHD	1.94	6.54	3.33	-7.57	-8.66	0.00	0.06
Apache Storm	9.97	15.70	12.26	3.20	5.40	0.98	1.00
Syncope	1.45	3.37	2.05	0.85	1.43	0.96	0.98
SystemML	2.65	9.88	4.45	1.97	1.50	0.86	0.74
Tapestry 5	17.65	41.69	25.77	5.54	19.56	0.78	1.00
Tika	5.88	13.94	8.48	-4.18	0.60	0.06	0.59
TomEE	0.99	7.44	2.27	-1.03	-1.62	0.24	0.24
VCL	19.52	53.16	29.12	-5.06	28.40	0.30	1.00
VYSPER	1.50	25.27	4.55	6.61	0.98	0.93	0.57
Struts 2	12.48	25.72	16.72	0.50	4.75	0.54	0.82
Hadoop Yarn	5.38	8.37	6.78	-3.48	-1.41	0.00	0.09
ZooKeeper	16.45	30.69	21.79	1.21	-2.51	0.60	0.31

Table 31 – μ posterior distribution summary, 'specific-models', 'Medium Priority' results (55 projects).

	Medium Priority (b)						
Project	CI_L	CI_U	μ_{MAP}	E[F1(b,a)]	E[F1(b,c)]	E[F2(b,a)]	E[F2(b,c)]
Buildr	5.96	14.90	9.26	-10.09	1.33	0.09	0.67
Camel	1.55	1.92	1.73	-0.34	-0.91	0.04	0.08
Cassandra	6.20	7.31	6.75	1.85	2.15	1.00	1.00
Commons Codec	3.11	24.96	7.77	2.96	7.70	0.68	0.93
Commons Collections	8.46	29.03	14.70	-7.17	-18.22	0.35	0.24
Commons Compress	3.09	6.61	4.48	-3.12	-17.30	0.13	0.02
Crunch	1.38	2.31	1.73	0.28	0.74	0.73	0.91
Derby	15.93	22.27	18.81	5.22	-3.64	0.97	0.35
Directory Kerberos	3.08	7.92	4.73	2.40	-20.71	0.89	0.09
MINA	10.85	32.85	18.31	14.55	6.76	0.99	0.80
FORTRESS	5.16	21.35	9.57	9.20	6.49	0.99	0.88
Flink	5.89	7.29	6.50	0.86	-2.11	0.91	0.00
FtpServer	3.27	11.96	5.82	0.28	-	0.60	-
Giraph	3.72	6.51	4.75	-1.27	1.68	0.27	0.84
Hadoop Core	6.92	8.66	7.72	1.25	0.66	0.94	0.78
HBase	3.78	4.36	4.06	0.84	-1.62	1.00	0.00
Hadoop HDFS	7.87	9.62	8.62	4.11	-0.54	1.00	0.29
Apache Helix	7.50	17.57	10.76	1.66	-10.00	0.67	0.13
Hive	6.43	7.13	6.77	-1.28	-0.32	0.01	0.26
Ignite	7.62	9.35	8.36	-2.39	2.92	0.06	1.00
Commons IO	5.91	28.97	11.67	3.04	1.21	0.67	0.60
Isis	3.34	7.84	5.07	-1.80	0.32	0.16	0.60
Ivy	12.70	32.89	19.08	9.91	-2.86	0.93	0.50
jclouds	8.41	14.28	11.20	-0.45	-0.53	0.49	0.49
Kafka	10.68	13.63	11.97	-5.24	6.04	0.01	1.00
Commons Lang	7.32	16.87	10.95	-0.08	9.86	0.51	1.00
Libcloud	2.98	6.58	4.45	-0.10	-0.25	0.51	0.57
Log4j 2	5.55	8.67	6.97	2.61	-5.09	0.96	0.05
Lucene	2.02	2.62	2.28	-1.49	-1.04	0.00	0.09
Apache MADlib	7.88	26.14	12.89	0.84	5.49	0.55	0.78
Mahout	5.77	10.12	7.44	2.86	2.32	0.97	0.85
Hadoop Map/Reduce	13.61	17.84	15.46	-2.09	7.71	0.21	1.00
Commons Math	4.54	8.72	6.37	2.90	1.83	0.98	0.79
Mesos	20.56	26.22	23.07	-6.98	14.96	0.06	1.00
Maven	3.90	7.03	5.17	-3.26	-9.38	0.13	0.01
Archiva	14.01	29.85	20.08	9.31	2.45	0.94	0.66
MyFaces	4.04	6.04	4.94	-0.43	2.60	0.41	1.00
Nutch	8.89	16.89	11.59	-2.55	-2.21	0.22	0.33
Oozie	20.11	26.27	22.72	11.84	13.01	1.00	1.00
OpenJPA	11.26	18.30	13.82	-1.04	-24.12	0.42	0.01
Phoenix	5.22	6.73	5.81	-0.73	0.86	0.31	0.81
Solr	8.89	11.96	10.18	2.61	1.79	0.99	0.87
Spark	5.29	6.09	5.67	0.97	1.57	1.00	1.00
MINA SSHD	7.80	15.89	10.55	7.57	-1.10	1.00	0.51
Apache Storm	8.12	10.85	9.37	-3.20	2.20	0.02	0.97
Syncope	1.15	1.77	1.38	-0.85	0.58	0.04	0.91
SystemML	2.77	4.34	3.37	-1.97	-0.46	0.14	0.44
Tapestry 5	16.68	29.95	21.73	-5.54	14.03	0.22	1.00
Tika	10.49	16.90	13.40	4.18	4.78	0.94	0.92
TomEE	2.82	5.98	3.95	1.03	-0.58	0.76	0.43
VCL	27.43	52.45	35.76	5.06	33.47	0.70	1.00
VYSPER	0.63	3.59	1.27	-6.61	-5.62	0.07	0.13
Struts 2	13.53	23.02	17.34	-0.50	4.24	0.46	0.82
Hadoop Yarn	9.13	11.42	10.09	3.48	2.07	1.00	0.99
ZooKeeper	17.64	26.38	21.16	-1.21	-3.73	0.40	0.19

Table 32 – μ posterior distribution summary, ‘specific-models’, ‘High Priority’ results (55 projects).

	High Priority (c)						
Project	CI_L	CI_U	μ_{MAP}	E[F1(c,a)]	E[F1(c,b)]	E[F2(c,a)]	E[F2(c,b)]
Buildr	1.76	24.78	5.02	-11.41	-1.33	0.10	0.33
Camel	1.54	4.33	2.36	0.56	0.91	0.77	0.92
Cassandra	3.60	5.77	4.46	-0.30	-2.15	0.30	0.00
Commons Codec	0.20	12.01	1.24	-4.74	-7.70	0.18	0.07
Commons Collections	3.90	104.72	19.23	11.07	18.22	0.63	0.75
Commons Compress	5.81	49.07	16.28	14.18	17.30	0.92	0.98
Crunch	0.39	2.38	0.82	-0.46	-0.74	0.23	0.09
Derby	11.26	40.30	18.78	8.86	3.64	0.89	0.65
Directory Kerberos	2.88	84.70	13.07	23.09	20.71	0.95	0.91
MINA	3.82	32.00	9.10	7.77	-6.76	0.87	0.20
FORTRESS	0.42	20.13	1.95	2.70	-6.49	0.68	0.12
Flink	7.46	10.00	8.64	2.97	2.11	1.00	1.00
FtpServer	-	-	-	-	-	-	-
Giraph	1.00	8.22	2.30	-2.95	-1.68	0.11	0.16
Hadoop Core	5.77	8.62	6.84	0.59	-0.66	0.73	0.22
HBase	5.01	6.43	5.63	2.46	1.62	1.00	1.00
Hadoop HDFS	7.72	10.93	9.06	4.65	0.54	1.00	0.71
Apache Helix	8.45	44.00	18.49	11.66	10.00	0.88	0.87
Hive	6.23	8.03	7.09	-0.96	0.32	0.10	0.74
Ignite	4.68	6.47	5.50	-5.31	-2.92	0.00	0.00
Commons IO	1.97	40.31	7.44	1.83	-1.21	0.53	0.39
Isis	1.93	10.45	3.88	-2.12	-0.32	0.18	0.40
Ivy	5.44	63.80	17.49	12.76	2.86	0.81	0.50
jclouds	5.59	21.35	9.57	0.08	0.53	0.50	0.50
Kafka	4.95	7.30	5.96	-11.27	-6.04	0.00	0.00
Commons Lang	0.45	3.80	1.03	-9.94	-9.86	0.00	0.00
Libcloud	0.88	14.46	2.76	0.16	0.25	0.44	0.42
Log4j 2	6.64	20.14	10.79	7.70	5.09	0.99	0.95
Lucene	2.02	5.22	3.04	-0.45	1.04	0.29	0.91
Apache MADlib	0.55	43.61	3.64	-4.65	-5.49	0.22	0.21
Mahout	2.53	10.15	4.36	0.53	-2.32	0.56	0.15
Hadoop Map/Reduce	6.52	9.47	7.74	-9.80	-7.71	0.00	0.00
Commons Math	1.25	11.98	2.98	1.07	-1.83	0.60	0.21
Mesos	6.61	10.25	8.18	-21.94	-14.96	0.00	0.00
Maven	6.79	27.78	12.64	6.12	9.38	0.85	0.99
Archiva	8.81	34.45	15.43	6.86	-2.45	0.82	0.34
MyFaces	1.35	3.94	2.09	-3.03	-2.60	0.01	0.00
Nutch	8.30	23.84	13.19	-0.34	2.21	0.45	0.67
Oozie	6.71	14.64	9.49	-1.18	-13.01	0.33	0.00
OpenJPA	19.23	68.84	34.33	23.09	24.12	0.98	0.99
Phoenix	3.47	7.18	4.98	-1.59	-0.86	0.16	0.19
Solr	6.27	11.52	8.06	0.82	-1.79	0.69	0.13
Spark	3.68	4.58	4.09	-0.61	-1.57	0.04	0.00
MINA SSHD	3.45	31.50	8.37	8.66	1.10	0.94	0.49
Apache Storm	5.71	9.02	7.01	-5.40	-2.20	0.00	0.03
Syncope	0.34	1.82	0.70	-1.43	-0.58	0.02	0.09
SystemML	1.68	7.94	3.42	-1.50	0.46	0.26	0.56
Tapestry 5	4.32	15.14	7.33	-19.56	-14.03	0.00	0.00
Tika	4.22	15.48	7.73	-0.60	-4.78	0.41	0.08
TomEE	2.09	9.41	4.10	1.62	0.58	0.76	0.57
VCL	0.90	13.88	2.89	-28.40	-33.47	0.00	0.00
VYSPER	0.89	27.43	3.49	-0.98	5.62	0.43	0.86
Struts 2	7.18	23.31	12.06	-4.75	-4.24	0.18	0.18
Hadoop Yarn	6.87	9.56	8.04	1.41	-2.07	0.91	0.01
ZooKeeper	18.97	33.31	24.73	2.51	3.73	0.69	0.80

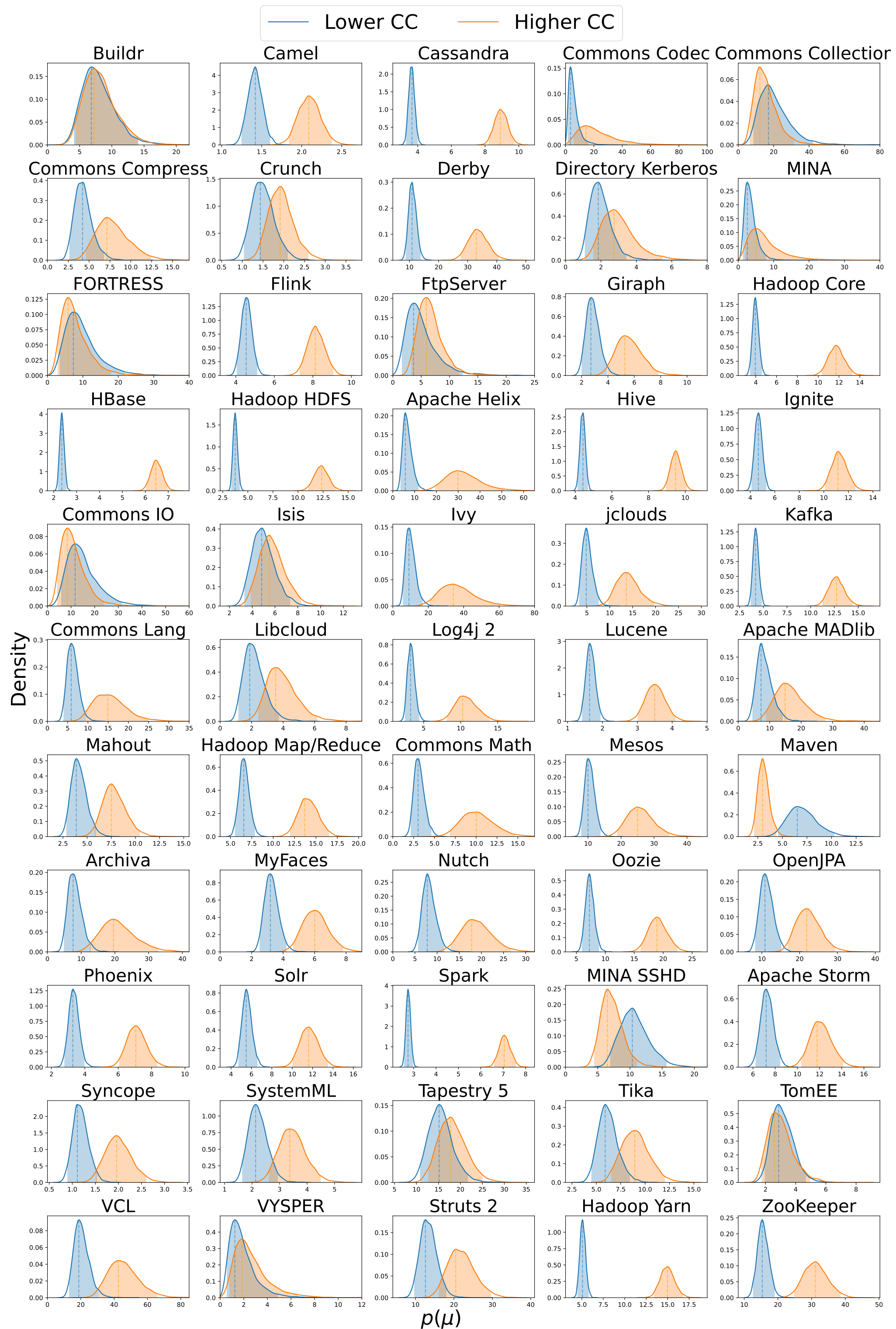


Figure 27 – μ posterior distributions - ‘specific-models’, ‘code-churn’ results, for all 55 projects. The average bug fixing time of reports lower code-churn values vs. higher code-churn values. The results diverge depending on the selected project, but there is a tendency that higher code-churn values patches demand more time to be fixed.

Table 33 – μ posterior distribution summary, 'specific-models', 'Lower Code-Churn' results (55 projects).

Lower Code-Churn (a)					
Project	CI_L	CI_U	μ_{MAP}	E[F1(a,b)]	E[F2(a,b)]
Buildr	4.10	14.08	6.82	-0.26	0.47
Camel	1.25	1.61	1.42	-0.68	0.00
Cassandra	3.32	4.01	3.66	-5.30	0.00
Commons Codec	1.42	16.40	3.53	-16.84	0.07
Commons Collections	8.31	42.58	17.09	4.76	0.67
Commons Compress	2.58	6.80	4.21	-3.55	0.05
Crunch	1.03	2.11	1.43	-0.44	0.14
Derby	8.66	13.90	10.88	-22.59	0.00
Directory Kerberos	1.06	3.47	1.85	-1.02	0.18
MINA	1.22	7.96	2.57	-4.05	0.19
FORTRESS	3.28	22.01	7.33	1.87	0.62
Flink	4.04	5.12	4.54	-3.60	0.00
FtpServer	1.49	12.33	3.65	-1.52	0.29
Giraph	2.00	3.97	2.71	-2.78	0.00
Hadoop Core	3.43	4.59	3.95	-7.77	0.00
HBase	2.18	2.55	2.35	-4.13	0.00
Hadoop HDFS	3.34	4.21	3.73	-8.58	0.00
Apache Helix	3.48	11.84	5.71	-25.61	0.00
Hive	4.13	4.70	4.42	-5.09	0.00
Ignite	4.09	5.32	4.66	-6.60	0.00
Commons IO	5.70	31.91	11.72	3.93	0.68
Isis	3.29	7.39	4.83	-0.76	0.31
Ivy	5.64	17.00	9.10	-26.55	0.00
jclouds	3.29	7.64	4.86	-8.91	0.00
Kafka	3.66	4.85	4.19	-8.50	0.00
Commons Lang	3.95	9.39	5.87	-9.24	0.00
Libcloud	1.14	3.75	1.88	-1.75	0.06
Log4j 2	2.47	4.42	3.23	-7.44	0.00
Lucene	1.41	1.95	1.63	-1.86	0.00
Apache MADlib	4.42	14.20	7.25	-8.14	0.06
Mahout	2.82	5.95	3.88	-3.66	0.01
Hadoop Map/Reduce	5.50	7.89	6.56	-7.51	0.00
Commons Math	2.12	4.64	3.03	-7.02	0.00
Mesos	7.89	13.82	10.00	-15.36	0.00
Maven	4.56	10.30	6.52	3.95	1.00
Archiva	4.84	12.97	7.61	-12.90	0.00
MyFaces	2.50	4.23	3.19	-2.80	0.00
Nutch	5.64	11.61	7.77	-10.75	0.00
Oozie	5.96	8.92	7.26	-11.76	0.00
OpenJPA	8.17	15.23	10.82	-11.11	0.00
Phoenix	2.77	3.98	3.28	-3.76	0.00
Solr	4.70	6.61	5.48	-6.10	0.00
Spark	2.57	2.97	2.75	-4.31	0.00
MINA SSHD	6.89	15.72	10.37	3.48	0.89
Apache Storm	6.22	8.47	7.19	-4.85	0.00
Syncope	0.88	1.54	1.11	-0.82	0.01
SystemML	1.63	2.94	2.13	-1.25	0.02
Tapestry 5	10.72	21.75	15.16	-2.66	0.27
Tika	4.48	8.44	5.92	-2.83	0.05
TomEE	1.93	4.71	2.87	0.12	0.56
VCL	12.94	30.89	18.81	-25.15	0.00
VYSPER	0.53	4.95	1.25	-0.77	0.33
Struts 2	9.62	18.10	12.59	-8.58	0.02
Hadoop Yarn	4.49	5.79	5.08	-9.86	0.00
ZooKeeper	12.60	19.12	15.36	-15.58	0.00

Table 34 – μ posterior distribution summary, 'specific-models', 'Higher Code-Churn' results (55 projects).

Higher Code-Churn (b)					
Project	CI_L	CI_U	μ_{MAP}	E[F1(b,a)]	E[F2(b,a)]
Buildr	4.24	14.28	7.24	0.26	0.53
Camel	1.84	2.38	2.09	0.68	1.00
Cassandra	8.21	9.73	8.90	5.30	1.00
Commons Codec	5.03	60.60	14.45	16.84	0.92
Commons Collections	6.39	31.52	12.23	-4.76	0.33
Commons Compress	4.60	12.44	7.13	3.55	0.95
Crunch	1.42	2.60	1.91	0.44	0.86
Derby	27.48	40.75	32.97	22.59	1.00
Directory Kerberos	1.54	5.47	2.74	1.02	0.82
MINA	1.80	20.85	5.03	4.05	0.80
FORTRESS	2.62	18.14	5.84	-1.87	0.38
Flink	7.31	9.07	8.13	3.60	1.00
FtpServer	3.26	11.82	5.92	1.52	0.71
Giraph	3.95	7.85	5.28	2.78	1.00
Hadoop Core	10.31	13.30	11.73	7.77	1.00
HBase	6.02	6.99	6.46	4.13	1.00
Hadoop HDFS	10.99	13.78	12.36	8.58	1.00
Apache Helix	19.30	51.11	29.86	25.61	1.00
Hive	8.93	10.09	9.47	5.09	1.00
Ignite	10.07	12.59	11.15	6.60	1.00
Commons IO	3.85	24.31	8.39	-3.93	0.31
Isis	3.82	8.32	5.50	0.76	0.69
Ivy	20.32	61.40	34.15	26.55	1.00
jclouds	9.68	19.85	13.62	8.91	1.00
Kafka	11.19	14.45	12.73	8.50	1.00
Commons Lang	9.13	25.06	14.91	9.24	0.99
Libcloud	2.39	6.22	3.53	1.75	0.94
Log4j 2	8.12	13.88	10.25	7.44	1.00
Lucene	3.00	4.10	3.50	1.86	1.00
Apache MADlib	8.65	27.71	14.89	8.14	0.94
Mahout	5.73	10.52	7.51	3.66	0.99
Hadoop Map/Reduce	11.96	16.56	13.68	7.51	1.00
Commons Math	6.79	14.65	10.02	7.02	1.00
Mesos	18.79	34.79	25.02	15.36	1.00
Maven	2.10	4.40	2.98	-3.95	0.00
Archiva	12.39	33.04	19.59	12.90	1.00
MyFaces	4.62	7.80	6.01	2.80	1.00
Nutch	12.93	26.80	17.80	10.75	1.00
Oozie	16.05	22.52	18.91	11.76	1.00
OpenJPA	16.80	29.51	21.83	11.11	1.00
Phoenix	6.03	8.36	7.06	3.76	1.00
Solr	10.02	13.53	11.60	6.10	1.00
Spark	6.58	7.58	7.05	4.31	1.00
MINA SSHD	4.40	11.12	6.49	-3.48	0.11
Apache Storm	10.32	14.16	11.78	4.85	1.00
Syncope	1.50	2.61	1.96	0.82	0.99
SystemML	2.61	4.51	3.37	1.25	0.98
Tapestry 5	12.60	25.13	17.77	2.66	0.73
Tika	6.51	12.09	8.94	2.83	0.95
TomEE	1.71	4.98	2.59	-0.12	0.44
VCL	30.54	65.96	42.58	25.15	1.00
VYSPER	0.78	6.62	1.86	0.77	0.67
Struts 2	15.93	29.71	20.48	8.58	0.98
Hadoop Yarn	13.42	16.76	14.98	9.86	1.00
ZooKeeper	24.70	38.66	31.11	15.58	1.00

APPENDIX C – PREDICTIVE CHECK

In this appendix, we provide two predictive checkings: on the prior and on the posterior distribution. The analysis objective is to ensure that the prior provides plausible values for BFT and that the posterior predictive provides generated data approximated to the original data.

Prior Predictive Check

The Fig. 28 shows the generated data for days (in a log scale). Both intervals are based on previous work (WEISS *et al.*, 2007; AKBARINASAJI *et al.*, 2018; SAHA *et al.*, 2014) that suggests that bugs are generally fixed in a few days (two or three), while others can take months or even years to be considered fixed. The prior predictive and their CI show a broad enough to contemplate both scenarios.

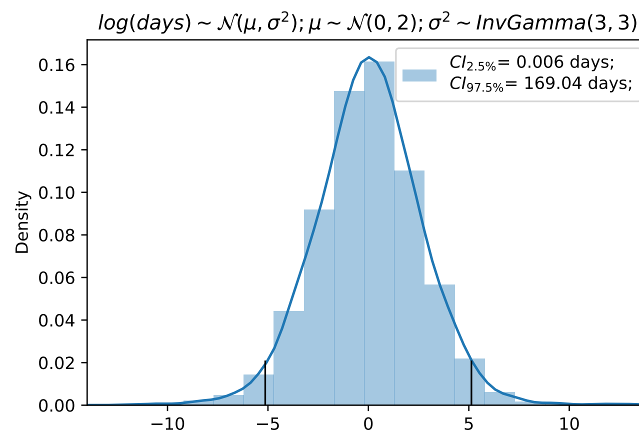


Figure 28 – Prior predictive. Data generated using the μ and σ^2 priors distributions. The prior (in a log-scale) is broad enough to provide values between 0.006 and 169 days.

Posteriori Predictive Check

Fig. 29 shows the generated data for days of the ‘specific-model’ applied for each of the 55 analyzed projects. The visualizations provide the model generated data -$y^\star$ - and the projects’ original data - y. While the original data follows a bimodal distribution in some cases, the predictive seems to contemplate most of the project’s data behavior, indicating a good fit to the project’s data.

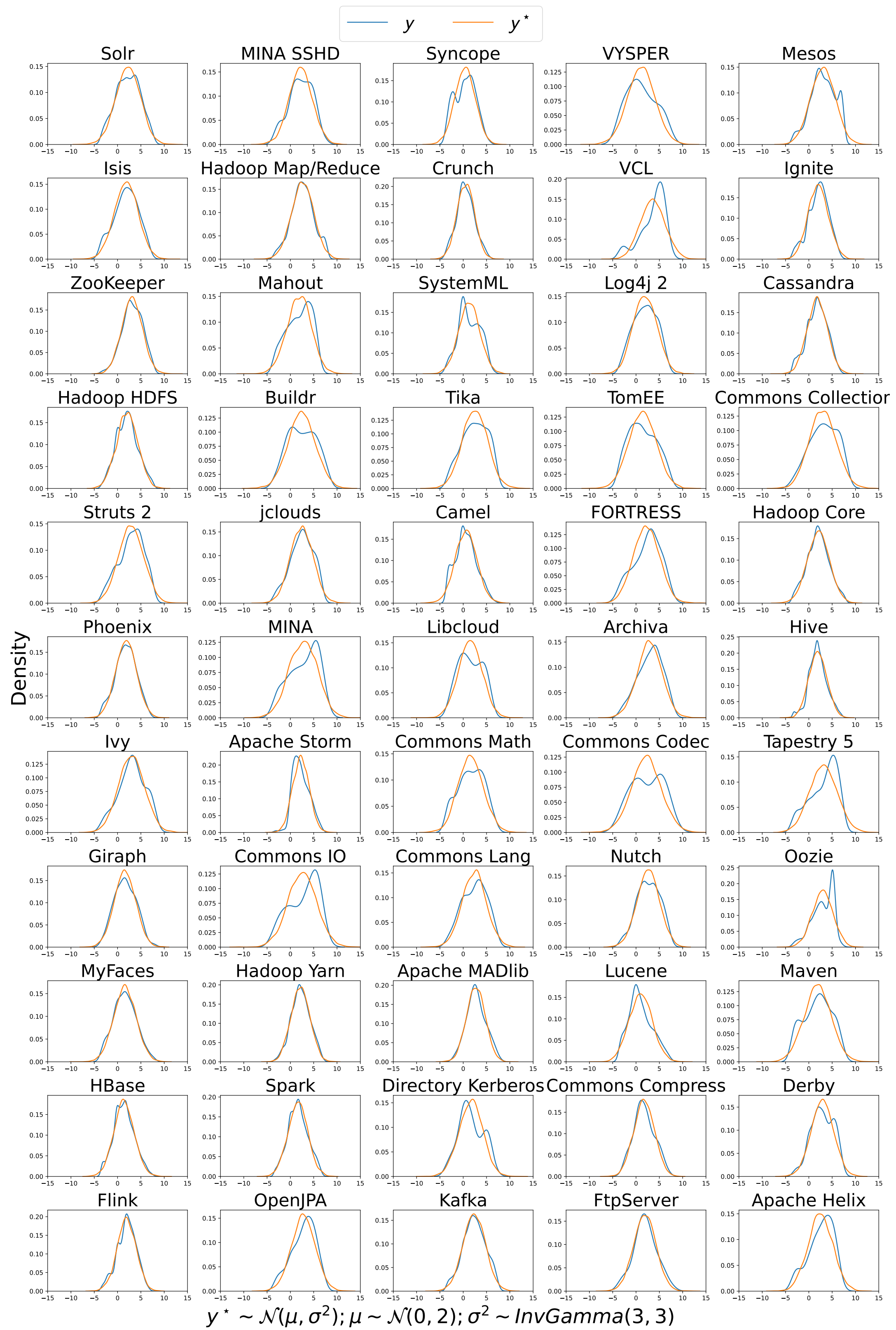


Figure 29 – Posterior predictive. We notice that the model-generated data distribution ($y^\star$, in orange) is very close to the original data distribution (y, in blue) for most projects.

APPENDIX D – ALTERNATIVE HIERARCHICAL MODELS

This appendix presents the alternative hierarchical models for 'priority' and 'code-churn' data. We already show the results for 'link' in section 4.5, where we also describe the models.

Priority

The Fig. 30 shows the three alternative hierarchical models μ_0 posterior distributions using the 'priority' data. Table 35 presents the posterior distributions summarization.

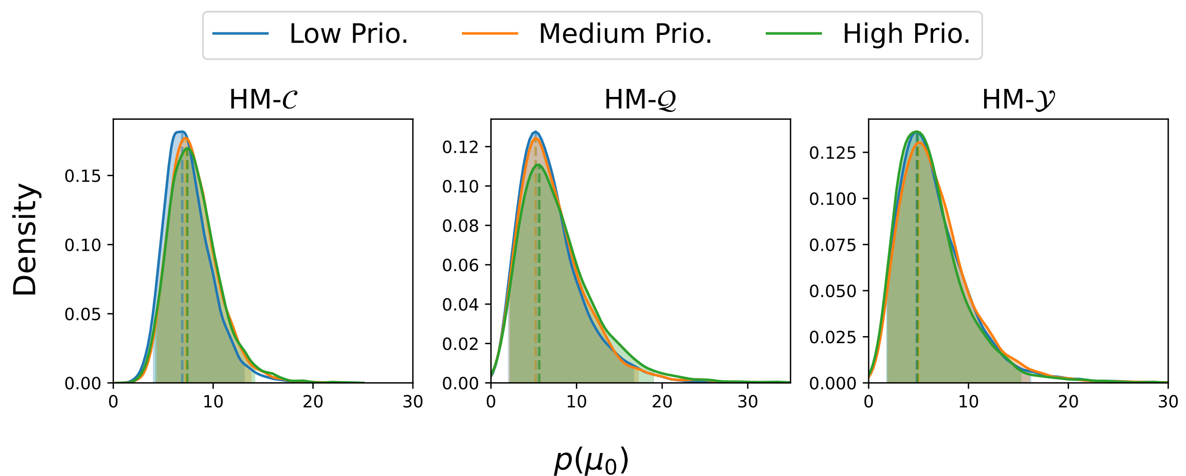


Figure 30 – μ_0 posterior distributions - 'HM-$\mathcal{G}$', 'priority' results

All models indicate no significant difference between the priority groups. For example, the average difference between the groups is always smaller than 1 (E[F1] in Table 35), and the probability of one group having a greater BFT than another is always close de 50%.

Code-Churn

The Fig. 31 shows the three alternative hierarchical models μ_0 posterior distributions using the 'code-churn' data. Table 36 presents the posterior distributions summarization.

All models indicate that bugs with higher code-churn values take more time to fix than those with smaller ones. However, there is more uncertainty about the difference between the groups compared to the 'HM-AP' model results, presented in Fig. 22 and Table 21. The average difference between the groups is always greater than 3 (E[F1] in Table 36). The

Table 35 – μ_0 posterior distribution summary from alternative hierarchical models (HM-$\mathcal{G}$), 'priority' results.

Low Priority (a)

Model	CI_L	CI_U	$\mu_{0_{MAP}}$	E[F1(a,b)]	E[F1(a,c)]	E[F2(a,b)]	E[F2(a,c)]
HM-$\mathcal{C}$	3.85	13.22	6.90	-0.53	-0.60	0.43	0.43
HM-$\mathcal{Q}$	1.84	16.93	5.22	-0.20	-0.78	0.48	0.45
HM-$\mathcal{Y}$	1.75	16.37	4.82	-0.22	0.26	0.48	0.52

Medium Priority (b)

Model	CI_L	CI_U	$\mu_{0_{MAP}}$	E[F1(b,a)]	E[F1(b,c)]	E[F2(b,a)]	E[F2(b,c)]
HM-$\mathcal{C}$	4.25	13.84	7.27	0.53	-0.07	0.56	0.49
HM-$\mathcal{Q}$	1.85	17.56	5.20	0.20	-0.58	0.51	0.47
HM-$\mathcal{Y}$	1.80	16.25	4.97	0.22	0.48	0.52	0.54

High Priority (c)

Model	CI_L	CI_U	$\mu_{0_{MAP}}$	E[F1(c,a)]	E[F1(c,b)]	E[F2(c,a)]	E[F2(c,b)]
HM-$\mathcal{C}$	4.14	14.25	7.40	0.60	0.07	0.57	0.50
HM-$\mathcal{Q}$	1.95	19.25	5.64	0.78	0.58	0.55	0.53
HM-$\mathcal{Y}$	1.63	15.48	4.90	-0.26	-0.48	0.48	0.46

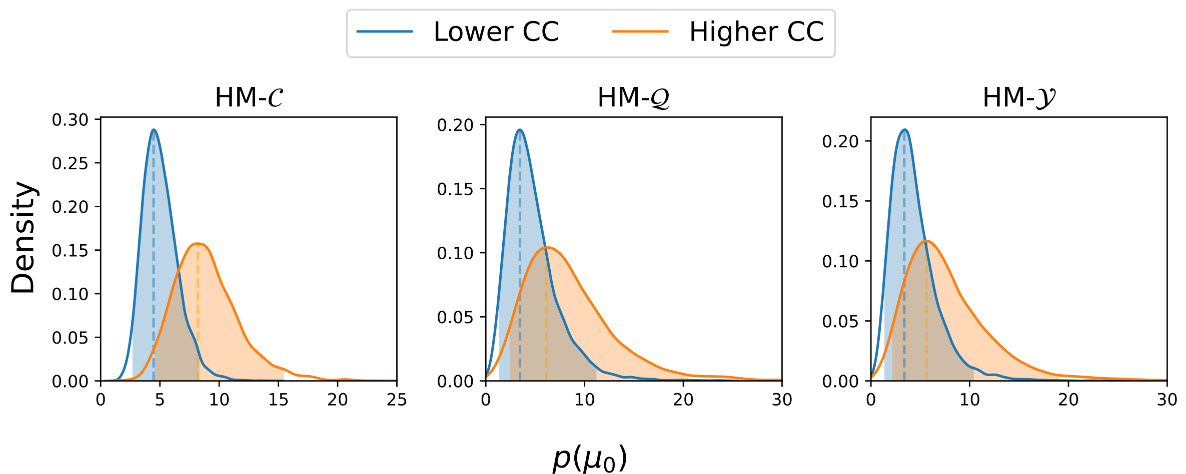


Figure 31 – μ_0 posterior distributions - 'HM-$\mathcal{G}$', 'code-churn' results.

Table 36 – μ_0 posterior distribution summary from alternative hierarchical models (HM-$\mathcal{G}$), 'code-churn' results.

Lower CC (a)

Model	CI_L	CI_U	$\mu_{0_{MAP}}$	E[F1(a,b)]	E[F2(a,b)]
HM-$\mathcal{C}$	2.68	8.40	4.47	-3.86	0.10
HM-$\mathcal{Q}$	1.26	11.21	3.46	-3.73	0.23
HM-$\mathcal{Y}$	1.20	10.49	3.37	-3.37	0.23

Higher CC (b)

Model	CI_L	CI_U	$\mu_{0_{MAP}}$	E[F1(b,a)]	E[F2(b,a)]
HM-$\mathcal{C}$	4.53	15.48	8.21	3.86	0.90
HM-$\mathcal{Q}$	2.18	20.77	6.12	3.73	0.77
HM-$\mathcal{Y}$	1.99	19.01	5.63	3.37	0.76

probabilities of associated higher code-churn bug patches demanding more time to be fixed than lower code-churn values patches are always greater than 70%.