Empirical Software Engineering (2022) 27:164
https://doi.org/10.1007/s10664-022-10213-7

The role of bug report evolution in reliable fixing estimation

Renan G. Vieira[1] · César Lincoln C. Mattos[1] · Lincoln S. Rocha[1] · João Paulo P. Gomes[1] · Matheus Paixão[2]

Accepted: 15 July 2022 / Published online: 20 September 2022

Abstract

Context Bug reports contain information that can be used by researchers and practitioners to better understand the bug fixing process and to enable the estimation of the effort necessary to fix bugs. In general, estimation models are built using the data (e.g., fixing time, severity, number of comments, number of attachments, and number of patches) present in the reports of fixed bugs (i.e., the report final's state). However, we claim that this approach is not reliable in a real setting. Effort estimation is necessary for bug fix scheduling and team allocation tasks, which happens closer to the bug report opening than its closing. At that moment, the data available in the bug report is less informative than the data used to build the model, which may lead to an unrealistic estimation.

Objective We propose a new approach to estimate bug-fixing time, i.e., the time span between the moment the bug was first reported until the bug is considered fixed. We consider not only the final state of the bug report to create our estimation model but all the previous available states, different from some previous studies that do not consider the reports' updates. The concept of bug report evolution is used to create a dataset containing all investigated report states.

Method First, we verify 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 distinct output configurations, and class balancing techniques. The experimental analysis is performed with data from the JIRA issue tracking system of ten open-source projects. By leveraging the best models (considering all possible configurations) for the different states of the evolution of a bug report, we can assess whether there are significant differences in the models' estimation ability due to the report's state.

Results We gathered evidence that the reports' fields are updated often, which characterizes the reports' evolution, impacting the building of bug-fixing estimation models. The models'

Communicated by: Foutse Khomh, Gemma Catolino, Pasquale Salza

This article belongs to the Topical Collection: *Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*

✉ Renan G. Vieira
renan.vieira@alu.ufc.br

Extended author information available on the last page of the article.

evaluation shows promising results 0.44 up to 0.85, precision values from 0.34 up to 0.74 and recall values from 0.62 up to 0.99, depending on the project.

Conclusions Our experiments show that field updates have a meaningful impact on the models' performance. Furthermore, we present a new approach to deal with the bug report evolution by considering each report version as an independent report. Finally, we also make available our dataset to the community.

Keywords Bug report · Machine learning · Effort estimation · Resolution time estimation · JIRA tracking issue system

1 Introduction

Open-source software is widely adopted by end-users and companies around the world (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. Software developers and engineers use several tools to improve the software development process. One of the most commonly used tool (Serrano and 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.

Bugs are 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). 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 ITSs 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).

Several researchers highlight the importance of being able to provide 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. In those cases, such estimations can be critical to their cost planning and release management. 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. 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 and Goseva-Popstojanova 2017). Thus, the prediction of a bug resolution time plays a significant role in project management, since it supports resource allocation and future release planning.

[1]An issue could represent a story, a bug, a task, or another issue type in the project.

For software that uses an ITS, bug identification is generally recorded in the ITS itself. Next, a bug triage happens, being mostly[2] a manual 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 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. However, the reported information and the decisions made in the triage step are error-prone. For instance, 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. In our previous study (Vieira et al. 2019), we identified several changes and additions to bug reports during their life cycle when analyzing bug reports from 55 open-source projects from Apache. For instance, we observed changes in a report's assignee on 54.63% of the bug reports, and description modifications on 18.16%, to mention a few.

Thus, it is evident that reports should not be seen or analyzed only when they are closed/resolved, where 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 zero 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 would be highly valuable. This paper investigates the viability of providing such a tool to help software managers while considering 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 investigate three main questions. 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 2.3) 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 balance strategies *ii*) the estimated label

[2]In Mozilla's (Firefox) case, it is partly automated, see https://hacks.mozilla.org/2019/04/teaching-machines-to-triage-firefox-bugs/

related to the resolution time window. To achieve that, we create a temporal dataset based on our previous work (Vieira et al. 2019) 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 2) the idea of bug reports evolution. To tackle this question, we look at every report state as an individual and independent report. After training the models with such an 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.

We organize the remainder of the paper as follows. Section 2 deals with the investigation methodology: dataset acquisition and creation, processing and description, and considered machine learning methods. Section 3 presents the evaluated models results and address our research questions. In Section 4 we draw the discussion regarding the results. Section 5 lists the threats to the validity of this work. In Section 6 we highlight and draw a comparison with selected related works. We conclude the paper in Section 7 with final thoughts and considerations regarding the research.

2 Materials and Methods

This section describes the materials and methods used to address our research questions. Sections 2.1 and 2.2 explain the dataset used in all the investigation steps. In the Section 2.3, we present the necessary information to answer RQ1. Sections 2.4 and 2.5 present the materials and methods to answer RQ2 and RQ3. We conclude this section describing in Section 2.6 the process to create the train/test data partition to train the machine learning models. This paper is associated with a replication package[3] with code, data, figures and tables.

Foremost, we want to clarify that we use 'Bug Report Resolution Time' and 'Bug-fixing time' as synonymous in the text. In this paper 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.

2.1 The JIRA Bug Report Dataset

In Vieira et al. (2019) we developed a dataset containing ten years of bug fixing activities (and reports) from 55 open source projects from the Apache ecosystem. In the following, we briefly describe the dataset. Additional details can be found in the original paper or the replication package.[4]

The dataset comprises JIRA bug reports of projects in different system categories, such as big-data, database, cloud, network-server, security, build-management, library, and machine-learning. All selected reports were created and resolved/closed between 2009 and 2018. Four categories of files were created: `changelog`, `commentlog`, `commitlog`,

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

[4] https://figshare.com/articles/Replication_Package_-_PROMISE_19/8852084

and `snapshot`. Each project has one file of each category, and each category presents its own set of unique attributes/fields. The `snapshot` files contain the reports as they are when resolved/closed. The `commentlog` is the record of all comments made during a report's life cycle. The `commitlog` files record the data related to the commits responsible for solving the bug report. The number of commits related to a certain bug report may vary. The maximum number of commits for a certain bug report in the collected dataset is 98, and the median is 1. The `changelog` files record all changes made in any of the reports' fields during its life cycle. In the Apache ecosystem, the JIRA issue tracking systems and the GitHub platform are the data sources (JIRA for bug reports records and GitHub for the commits related to these bug reports).

2.2 A Temporal Dataset of Bug-Fixing Activities and Reports

Using the dataset proposed in Vieira et al. (2019), 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 by Vieira et al. (2019), and the subset that we use in our experiments are listed in Section 2, Table 4.

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 **CD** be the report's Creation Date and let **RD** be the report's Resolution Date. The variable **RRT**, i.e. the Report Resolution Time, is defined as $\mathbf{RRT} = \mathbf{RD} - \mathbf{CD}$.

The dataset original structure proposed by Vieira et al. (2019), 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 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 paper, 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 **intermediate 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, as proposed by Vieira et al. (2019), 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. Figure 1 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 s) 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.

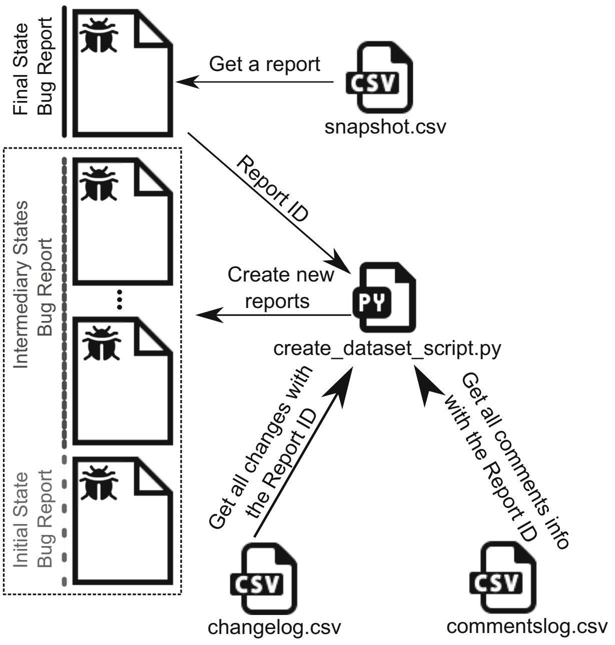


Fig. 1 The Temporal dataset of bug-fixing activities and reports creation process

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.

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) return temporal_dataset
```

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.

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. Figure 2 summarizes the definition.

2.3 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 6. 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.

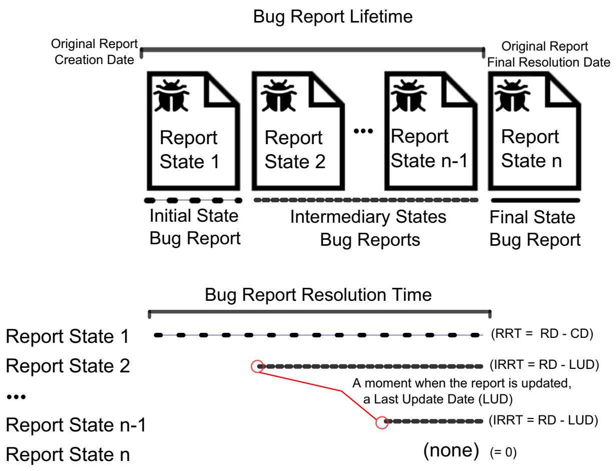


Fig. 2 Bug reports resolution time

The work developed by Zhang et al. (2013) uses a KNN (K-Nearest Neighbors algorithm) with a set of standard report's attributes. Table 1 lists their names, descriptions, and if it is present or not in the dataset proposed in our previous study (Vieira et al. 2019).

We only use a subset of the original attributes, as our dataset does not contain all of them, as shown in Table 1. 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

$$d_p(P_a, P_b) = |P_a - P_b| \times 0.2. \tag{1}$$

$$d_s(S_a, S_b) = 1 - \frac{|S_a \cap S_b|}{|S_a \cup S_b|}. \tag{2}$$

Both (1) and (2) are adaptations from the work by Zhang et al. (2013). Equation (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 (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 (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 2 presents the list of attributes for each *Set*.

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

Table 1 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

Table 2 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`

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. We want to verify how the information difference between initial and final reports impacts the models' performance with these two experiments. The hypothesis is that the last report state contains more information, and the models trained with them (EXP1) may provide better results than the models trained with the initial states reports (EXP2). 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 (the unique identifier number, the key provided by the Jira ITS when the bug is reported). We do not look at the report features at this data partition step. 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 Section 2.6.

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 h) 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 h) 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 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 Section 2.5). Thus, it will help us to compare the solutions (our approach and the baseline approach).

2.4 Preprocessing Steps on the 'Temporal Dataset' to Apply our Approach

For our approach, we choose to use the temporal dataset process creation on 10 of the 55 projects from the original dataset by Vieira et al. (2019), 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 3). This also increases the time to train the models, as we train different machine learning models in different configurations (to be explored in Section 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 3 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

Table 3 Filtered dataset information

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

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 purposes. 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 min, 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.

Table 3 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 by Vieira et al. (2019). In the third column, we have the number of reports after applying the creation process, as described in Section 2.2. 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 total numbers of states created from the Selected Reports.

The original `snapshot.csv` file contains 53 attributes, but we only use a subset of these alongside some attribute variations. Table 4 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 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 4; 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 4 (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.

Table 4 Dataset features description

Attribute name	Description	Possible values	Addition information
`NoAttachedPatches`	Number of patches attached to the report	$\mathbb{N}$	Same idea of the original
`NoAttachments`	Number of files attached to the report	$\mathbb{N}$	Same idea of the original
`NoComments`	Number of comments in the report	$\mathbb{N}$	Same idea of the original
`Priority`	Report priority label encoding	{1,2,3,4,5}, meaning, respectively, 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	$\mathbb{N}$	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	$\mathbb{N}$	Binary attribute derivated from the "Components" field in the original dataset.
`NoDescriptionTopWords`	Number of Top 1000 most frequent words of a bug detailed description	$\mathbb{N}$	A simplification of the original dataset "DescriptionTopWords" field.

Table 4 (continued)

Attribute name	Description	Possible values	Addition information
`UniqueNoDescriptionTopWords`	Number of unique Top 1000 most frequent words of a bug detailed description	$\mathbb{N}$	A simplification of the original dataset “DescriptionTopWords” field.
`NoSummaryTopWords`	Number of Top 1000 most frequent words of a brief one-line bug summary	$\mathbb{N}$	A simplification of the original dataset “SummaryTopWords” field.
`UniqueNoSummaryTopWords`	Number of unique Top 1000 most frequent words of a brief one-line bug summary	$\mathbb{N}$	A simplification of the original dataset “SummaryTopWords” field.
`NoCommentsTopWords`	Number of Top 1000 most frequent words of a bug detailed summary	$\mathbb{N}$	A simplification of the original dataset “CommentsTopWords” field.
`UniqueNoCommentsTopWords`	Number of Top 1000 most frequent words of a bug detailed summary	$\mathbb{N}$	A simplification of the original dataset “CommentsTopWords” field.
`TotalLinks`	The number of other issue reports linked to the report.	$\mathbb{N}$	A aggregation of the original dataset “InwardIssueLinks” & “OutwardIssueLinks” fields.

Table 4 (continued)

Attribute name	Description	Possible values	Addition information
`DSLU`	Days Since the Last report Update	$\mathbb{N}$	- 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	$\mathbb{N}$	- 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	$\mathbb{N}$	- Created on the temporal dataset process creation.
`Progress`	Actual report state divided by the number of report states	{0...1}	Used in results analysisd
`ResolutionTimeInDays`	The report resolution time in days.	$\mathbb{N}$	- Dependent variable that the models try to predict;

2.5 Models Training Methodology

We choose three machine learning methods to create models using the temporal dataset: logistic regression, deep MultiLayer Perceptron (MLP), and Gaussian process. For the logistic regression we use the sklearn.[5] For the Gaussian process model, we use the GPFlow[6] implementation. For the Deep MLP, we consider the Keras library.[7] All models were trained using a 5-fold data partition to perform cross-validation. More details on how we perform this partition in Section 2.6.

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, inf]; and "two labels (threshold = 10 days)", where the reports are grouped by in two intervals: [0, 10[, [10, 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 5.

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 'dnn' for deep neural networks (deep 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. OD means using the original data, SMOTE (Chawla et al. 2002) implies the use of oversampling of minority classes. 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.

[5]https://scikit-learn.org/
[6]https://www.gpflow.org/
[7]https://keras.io/

Table 5 Labels distribution

	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

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 deep 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). For more details about the training process, we refer the reader to the replication package, which contains all the models' information.

2.6 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.

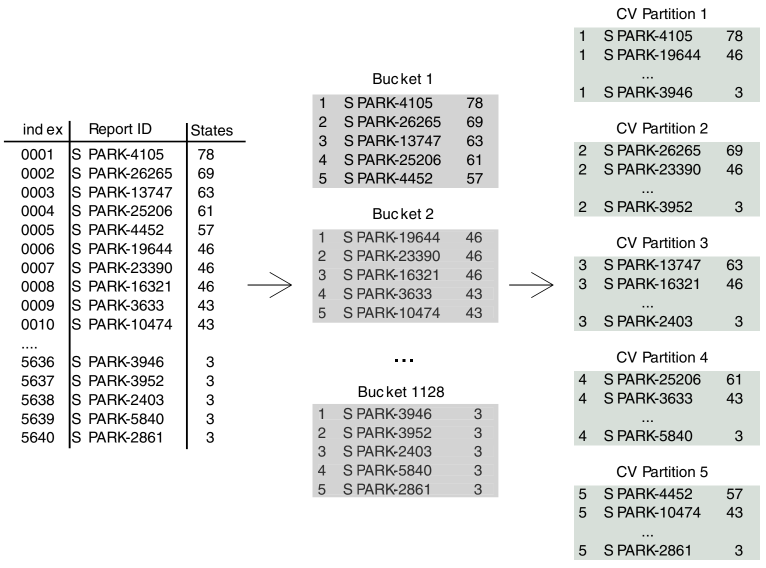
– Each bucket index a report by a number $k \in \{1, 2, 3, 4, 5\}$. The index of the report indicates the partition/fold where the report goes into.

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 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. 3, which contains a real values example of the process applied to the project Spark.

3 Results

3.1 Field Changes Analysis and Zhang et al. (2013) Replication (Baseline)

Table 6 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 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



index	Report ID	States
0001	S PARK-4105	78
0002	S PARK-26265	69
0003	S PARK-13747	63
0004	S PARK-25206	61
0005	S PARK-4452	57
0006	S PARK-19644	46
0007	S PARK-23390	46
0008	S PARK-16321	46
0009	S PARK-3633	43
0010	S PARK-10474	43
....		
5636	S PARK-3946	3
5637	S PARK-3952	3
5638	S PARK-2403	3
5639	S PARK-5840	3
5640	S PARK-2861	3

Bucket 1

1	S PARK-4105	78
2	S PARK-26265	69
3	S PARK-13747	63
4	S PARK-25206	61
5	S PARK-4452	57

Bucket 2

1	S PARK-19644	46
2	S PARK-23390	46
3	S PARK-16321	46
4	S PARK-3633	43
5	S PARK-10474	43

...

Bucket 1128

1	S PARK-3946	3
2	S PARK-3952	3
3	S PARK-2403	3
4	S PARK-5840	3
5	S PARK-2861	3

CV Partition 1

1	S PARK-4105	78
1	S PARK-19644	46
	...	
1	S PARK-3946	3

CV Partition 2

2	S PARK-26265	69
2	S PARK-23390	46
	...	
2	S PARK-3952	3

CV Partition 3

3	S PARK-13747	63
3	S PARK-16321	46
	...	
3	S PARK-2403	3

CV Partition 4

4	S PARK-25206	61
4	S PARK-3633	43
	...	
4	S PARK-5840	3

CV Partition 5

5	S PARK-4452	57
5	S PARK-10474	43
	...	
5	S PARK-2861	3

Fig. 3 The train/test 5-fold split method applied to project Spark

Table 6 Top bug reports' fields changes

	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%

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.

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 Section 2.3. Table 7 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 7 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 and f-measure 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 f-measure 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 8 shows the results using *Set 2* (see Table 2). The experiment uses the same data unit splits and data scenarios but with different attributes. The best results are highlighted in **boldface**.

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 cases of smaller unit values in Hadoop Core, where the EXP3 presents better results than EXP1 and EXP2, the values are close. 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?**

Table 7 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 8 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

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 f-measures absolute values, depending on the project and threshold values) than using the initial report states.

3.2 Training Models with All Bug Reports States

Table 9 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. The complete list of results with all 240 models configurations results can be found in the replication package. We evaluate the models using five metrics: log-loss (LOGLOSS), accuracy (ACC), f1-measure score (F1), precision, and recall. 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 balance 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 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. 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 f-measure due to the data imbalance nature (see Table 5).

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 $0.55 \sim 0.65$ and f-measure around $0.47 \sim 0.77$. 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.*

3.3 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[,

Table 9 All projects overall best results

Model	ACC	F1	Precision	Recall	LOGLOSS
flink_logreg_two_labels_5_CC	0.5737	**0.5800**	0.5026	0.6863	0.6608
flink_gp_two_labels_5_SMOTE	0.6139	0.5530	0.5519	0.5621	0.6513
flink_dnn_two_labels_5_SMOTE	0.5852	0.5536	0.5164	0.6028	0.6587
hadoop_logreg_two_labels_5_CC	0.6054	**0.5756**	0.5691	0.5828	0.6648
hadoop_gp_two_labels_5_CC	0.5748	0.5727	0.5394	0.6534	0.6793
hadoop_dnn_two_labels_5_OD	0.6053	0.5157	0.5891	0.4643	0.6617
hdfs_logreg_two_labels_5_RND	0.6124	0.5661	0.5592	0.5733	0.6532
hdfs_gp_two_labels_5_CC	0.5725	**0.5945**	0.5134	0.7161	0.6968
hdfs_dnn_two_labels_5_SMOTE	0.5857	0.5425	0.5280	0.5590	0.6524
lucene_logreg_two_labels_5_RND	0.6114	**0.4720**	0.3921	0.5956	0.6581
lucene_gp_two_labels_5_RND	0.3786	0.4562	0.3131	0.8897	0.7545
lucene_dnn_two_labels_5_CC	0.5123	0.3620	0.2933	0.4741	0.9287
mapreduce_logreg_two_labels_5_CC	0.6100	0.6053	0.5702	0.6455	0.6529
mapreduce_gp_two_labels_5_CC	0.5569	**0.6255**	0.5295	0.8259	0.6894
mapreduce_dnn_two_labels_5_SMOTE	0.5945	0.5876	0.5566	0.6258	0.8741
spark_logreg_two_labels_5_RND	0.6416	0.6228	0.5450	0.7269	0.6159
spark_gp_two_labels_5_CC	0.5776	**0.6284**	0.4897	0.8784	0.6708
spark_dnn_two_labels_5_SMOTE	0.6530	0.6075	0.5635	0.6600	0.6412
yarn_logreg_two_labels_5_CC	0.5929	0.5707	0.5554	0.5881	0.6626
yarn_gp_two_labels_5_CC	0.6069	0.5677	0.5768	0.5762	0.6604
yarn_dnn_two_labels_5_SMOTE	0.5861	**0.5832**	0.5456	0.6321	0.6426
zookeeper_logreg_two_labels_5_OD	0.6557	0.7723	0.6639	0.9266	0.6306
zookeeper_gp_two_labels_5_OD	0.6317	0.7731	0.6339	0.9920	0.7034
zookeeper_dnn_two_labels_5_OD	0.6589	**0.7764**	0.6626	0.9404	0.6258
kafka_logreg_two_labels_5_CC	0.6426	0.6643	0.6175	0.7192	0.6144
kafka_gp_two_labels_5_RND	0.6337	**0.6738**	0.6052	0.7756	0.6288
kafka_dnn_two_labels_5_SMOTE	0.6351	0.6552	0.6105	0.7165	0.6300
solr_logreg_two_labels_5_CC	0.6000	**0.6092**	0.5718	0.6530	0.6650
solr_gp_two_labels_5_RND	0.5717	0.5466	0.5910	0.5920	0.6834
solr_dnn_two_labels_5_SMOTE	0.5998	0.5943	0.5758	0.6173	0.6769

[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. Figure 4 summarizes the whole process. Figures 5 and 6 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 9, to perform the analysis.

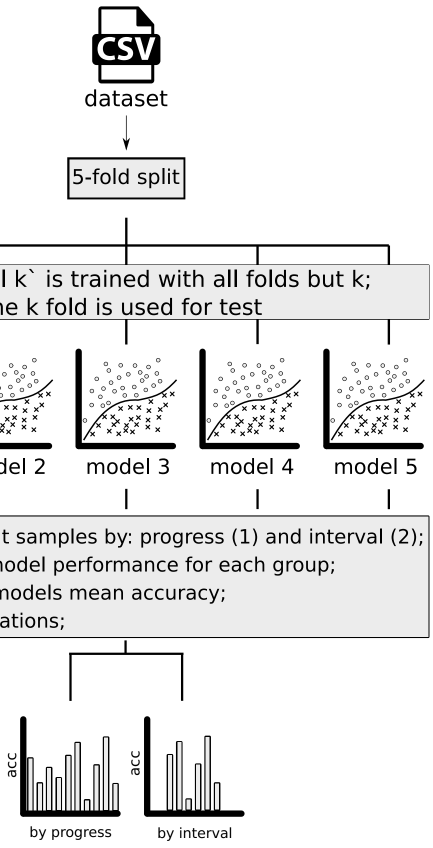

CSV
dataset
5-fold split
A `model k` is trained with all folds but k;
the k fold is used for test
model 1
model 2
model 3
model 4
model 5
1) Group test split samples by: progress (1) and interval (2);
2) Evaluate the model performance for each group;
3) Calculate the models mean accuracy;
4) Create visualizations;
acc
by progress
acc
by interval

Fig. 4 Workflow to reports RRT evaluation by progress and interval

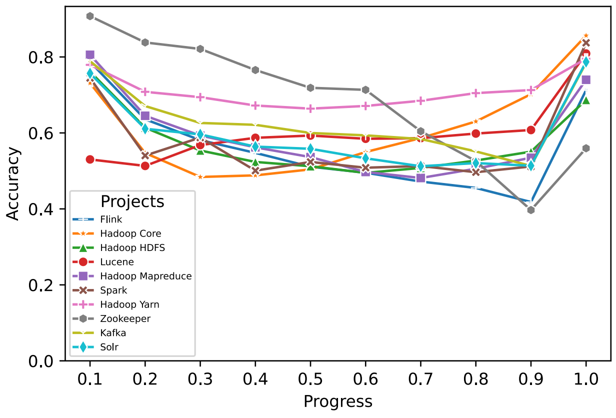


Fig. 5 Accuracy evaluation by report resolution progress

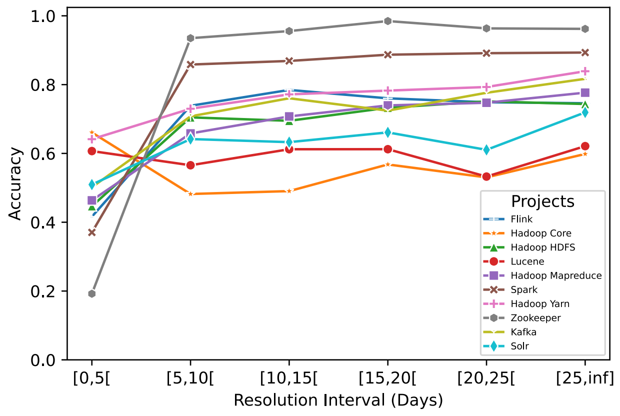


Fig. 6 Accuracy evaluation by report resolution interval

The Fig. 5 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. 5 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 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 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 10 shows the model's performance classifying all reports (same as the selected models at Table 9) 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 10 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 f-measure 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 8). 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.* The Fig. 6 shows the models' accuracy in a few RRT 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 more smother variation, with 'Hadoop Core' being more erratic as the interval values increase.

The results raise another question: why the performance drops in intermediate states? To explore this question, we need to set up some premises. The best models are the ones that classify bug reports that will be fixed in less - class 0 - or more - class 1—than five days (Table 9). Notice that all posterior states of an initial class 0 report will always be class 0—if its initial state is class 0, all its posterior states will also be class 0, as their RRT always decreases with each update, so they will never have RRT greater than five days. On the other hand, some initial reports of class 1 may have some of its posterior states as class 0. In fact, they have at least one state class 0 (the final state that always has RRT equal zero), and may have others as class 1, the others states besides the initial, that are a result of updates that occur prior five days of the report resolution date.

The first thing to notice is the precision suffers less with the addition of all states: the models perform very similarly considering all or only the initial states, to identify class 0 reports (sometimes the use of all states performs very similar or even better to identify correctly class 0 reports - Hadoop Core, Lucene, Spark, Solr, also in Table 10). Independent of looking only at initial states or all states, the models tend to classify class 0 reports as class 1 (Fig. 6 shows how bad the results are for report states with RRT smaller than five days, considerably worse than reports with reports RRT greater than 5). The hypothesis is

Table 10 Model 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

Table 10 (continued)

Project	Up to state	ACC	F1	Precision	Recall
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

that there is a similarity between different states of a same report, that when a class 0 initial report is classified as 1, its posteriors states (all of them are class 0) are very similar and are kept classified as 1, something like an error propagation from the initial report to the posterior ones. This is the first problem with the approach.

We did a performance analysis on the class 0 reports predictions to check the hypothesis around this first limitation. Given all initial class 0 reports were misclassified by the model as class 1, we calculate the percentage of the cases where all their posterior states also were misclassified as class 1. The results by project are: Zookeeper (78.2%), Flink (52.1%), Spark (49.7%), Kafka (35.0%), Hadoop MapReduce (33.7%), Hadoop HDFS (28.2%), Solr (26.5%), Lucene (13.5%), Hadoop Yarn (12.3%), Hadoop Core (5%). While the percentages vary for each project, it is important to notice that these are very extreme cases: all intermediate states are also misclassified. In another analysis, we calculate the percentage of these cases where at least half of the intermediate states are also misclassified, when the initial one was also wrongly classified as 1: Zookeeper (87.8%), Spark (81.3%), Flink (67.6%), Hadoop MapReduce (66.1%), Solr (57.4%), Hadoop HDFS (57%), Kafka (53.8%), Hadoop Yarn (40.3%), Lucene (33.5%), and Hadoop Core (32.2%). The presented results show that this is the case in at least one-third of reports of all projects. It is important to notice that the number of intermediate states is greater than the numbers of initial and final states reports (each report only has one initial and one final, but several intermediate). These numbers provide evidence that different states may not be so different from each other, and when one is misclassified, this error is propagated for the other similar states.

On the other hand, the models are very good at identifying the initial class 1 states reports (see the recall in Table 10, for most projects with values above 0.8 and 0.9). However, the metric values drop when we use all states, indicating that the models can identify the initial class 1 reports but not the intermediate class 1 reports. This is the second limitation with our approach, that comes from how the models are misclassifying intermediate states class 1 as class 0 reports. We evaluate how the number of report states impacts the model's performance on the class 1 reports. The argument here is that reports with several states may be more challenging to model, as their different states may be very similar (similar to the first limitation in class 0 reports). Also, the simple fact that they have a significant state number may provide more changes and give them an outlier behavior not present in reports with a smaller number of states. We perform an analysis where we calculate the metrics values in a particular way to identify the impact of the number of states. First, we group all states of the same report and then calculate the metrics at a report level. For example, given a report with ten states and the model classify 8 of them correctly, the accuracy is 0.8. We calculate all reports' metrics with this strategy and present the average in Table 11. With this performance at a report level, we split the reports' predictions into two groups: reports with higher and smaller numbers of states. The threshold is the project's median number of states. We also calculated the Spearman correlation coefficient between the accuracy of these groups and their number of states. Table 11 shows the accuracy (ACC), f1-score (F1), and precision (PRC) results, along with the correlation coefficient (in the last three columns) between the number of states and the metrics.

The results show a higher accuracy when classifying reports with a small number of states and a slight negative correlation between accuracy and the number of states of a report in 8 of 10 projects. However, the difference between the groups is not so evident and significant in the f-measure and precision results. While both analyses are not conclusive about the reason for the performance drop in intermediate states, both indicate that the excessive number of states could be an interesting exploration to mitigate the limitations of our proposed approach.

4 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

Table 11 Results comparison between groups of reports with more and fewer states

Project	More states			Fewer states			Correlation with number of states		
	ACC	F1	PRC	ACC	F1	PRC	ACC	F1	PRC
Flink	0.63	0.65	0.71	0.68	0.68	0.70	−0.1	−0.06	0.02
Hadoop Core	0.61	0.60	0.83	0.69	0.68	0.84	−0.18	−0.16	−0.07
Hadoop HDFS	0.64	0.68	0.73	0.69	0.71	0.72	−0.09	−0.06	0.04
Lucene	0.6	0.54	0.66	0.61	0.49	0.56	−0.01	0.11	0.12
Hadoop Mapreduce	0.66	0.68	0.73	0.69	0.71	0.72	−0.09	−0.06	0.04
Spark	0.77	0.80	0.80	0.82	0.83	0.84	−0.17	−0.14	−0.17
Hadoop Yarn	0.74	0.72	0.78	0.69	0.67	0.67	0.2	0.2	0.24
Zookeeper	0.74	0.81	0.75	0.72	0.77	0.69	0.00	0.04	0.07
Kafka	0.68	0.70	0.74	0.70	0.67	0.69	−0.04	0.04	0.01
Solr	0.65	0.66	0.79	0.71	0.68	0.75	−0.11	−0.00	−0.05

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 a 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 $0.5 \sim 0.7$), even though these are approximated values 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, 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 intermediate 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 could be different 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 revisit the analyses presented at the end of the Results section on why the performance drop in intermediate 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 a 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. It is also essential to notice that the analyses on the performance drop in intermediate states are performed in models trained with all states. We believe that a proper conclusion about the number of states' effects on the models' performance would only be achieved with a new round of training removing unnecessary states. However, removing some states without a proper characterization of what is useful or not in the modeling process would not provide sound conclusions. Therefore, we indent to characterize these types of updates and states in future works properly, as it seems a natural unfolding from the findings of this manuscript.

In this paper, 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.

5 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 a 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 a 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 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.

6 Related Works and Comparison

In this section, we discuss the related works and compare a few of them to our approach. 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 KNN model (Zhang et al. 2013), test several variations of the 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 0,6710, a smaller but comparable value than the best results present in Table 10. 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 and Weimer 2007; Kim and 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 the 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 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 MAE (Mean absolute error) 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 and 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% 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 k-NN-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 and 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 subject. 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, along with our paper's two significant novelty contributions compared with those presented in this section. The first one is the results regarding how the different states reports—initial and final states, explored in EXP1, EXP2, EXP3, and RQ1—impact the models' estimation reliability. To the best of our knowledge, this is the first paper to address this question explicitly. Our work also shows that training the models with unappropriated states (i.e., last state reports) can provide optimistic results when it is usually necessary to estimate bug-fixing time (i.e., initial state reports). The second one is the evaluation of a new approach to incorporate the reports updates dynamic into traditional machine learning methods. Although our work is not the first paper to provide this kind of insight—Habayeb et al. (2018) proposal estimates the RRT in different states as well, and Al-Zubaidi et al. (2017) shows some concern about the fields changes during feature selection —, our proposal relies upon a new idea of different reports states as snapshots to train the models. This allowed us to provide a more detailed analysis of how they perform in distinct moments of a report life cycle.

Another noticeable thing is the relatively small variation in machine learning methods on the majority of the papers, being most of them more traditional ones such as SVM, Random Forest, KNN, Logistic Regression, and Decision Tree. In our case, we aim to diversify our experiments with different types of models. For instance, we include neural networks models and Gaussian processes in our evaluations, choices that are rarely (if ever) seen in this kind of problem, maybe due to the usually small-sized datasets.

7 Conclusion

This paper investigates how the bug reports field updates impact the bug-fix time prediction using machine learning models. We use ten open-source projects from JIRA where we mine the bug reports data to train the models and draw our conclusions: Hadoop Core, Hadoop MapReduce, Hadoop HDFS, Hadoop Yarn, Lucene, Kafka, Solr, Zookeeper, Flink, and Spark. 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 Solr). The neural networks provide the best results for two projects, Hadoop Yarn and Zookeeper. For all projects, except Zookeeper, the use a data balance strategy improve the final results in all in selected models: undersampling for the Logistic Regression and Gaussian Process, and the oversampling for the neural network for Hadoop Yarn data.

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 their creation moment until their 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 have an impact on the models' performance. We get the best results when predicting the resolution time at the initial report states (close to data creation), which is 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 are also comparable to similar works with different data. All selected attributes are easy to compute and understand, ideal for a real-world use scenario.

For future works, we want to test more attributes that use information from previous reports states. Our approach considers each report as an independent pattern. The addition of more evolutional context in the attributes could improve the results. We only tested a simple natural language processing technique (BoW) on textual attributes, and more complex ones might provide better results. We want to categorize these corrected reports and create a profile that indicates what makes them easier to classify. After improving the results, feature importance could be used to define the more relevant fields in the report that helps to provide consistent RRT estimators. This could lead us to best practices when creating new bug reports.

References

Akbarinasaji S, Caglayan B, Bener A (2018) Predicting bug-fixing time: a replication study using an open source software project. J Syst Softw 136:173–186. https://doi.org/10.1016/j.jss.2017.02.021. http://www.sciencedirect.com/science/article/pii/S0164121217300365

Al-Zubaidi WHA, Dam HK, Ghose A, Li X (2017) 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, PROMISE. Association for Computing Machinery, New York, pp 53–62. https://doi.org/10.1145/3127005.3127011

Ardimento P, Bilancia M, Monopoli S (2016) Predicting bug-fix time: using standard versus topic-based text categorization techniques. In: Calders T, Ceci M, Malerba D (eds) Discovery science. Springer International Publishing, Cham, pp 167–182

Assar S, Borg M, Pfahl D (2016) Using text clustering to predict defect resolution time: a conceptual replication and an evaluation of prediction accuracy. Empirical Softw Engg 21(4):1437–1475. https://doi.org/10.1007/s10664-015-9391-7

Baysal O, Holmes R, Godfrey MW (2013) Situational awareness: personalizing issue tracking systems. In: Proceedings of the 2013 international conference on software engineering, ICSE ’13. IEEE Press, pp 1185–1188

Bhattacharya P, Neamtiu I (2011) Bug-fix time prediction models: can we do better? In: Proceedings of the 8th working conference on mining software repositories, MSR ’11. Association for Computing Machinery, New York, pp 207–210. https://doi.org/10.1145/1985441.1985472

Brady F (2013) Cambridge university report on cost of software faults, press release. http://www.prweb.com/releases/2013/1/prweb10298185.htm. Accessed 2020-01-02

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) Smote: synthetic minority over-sampling technique. J Artif Intell Res 16:321–357

Ebrahimi N, Trabelsi A, Islam MS, Hamou-Lhadj A, Khanmohammadi K (2019) An hmm-based approach for automatic detection and classification of duplicate bug reports. Inf Softw Technol 113:98–109. https://doi.org/10.1016/j.infsof.2019.05.007. http://www.sciencedirect.com/science/article/pii/S095058491930117X

Guo PJ, Zimmermann T, Nagappan N, Murphy B (2011) “Not my bug!” and other reasons for software bug report reassignments. In: Proceedings of the ACM 2011 conference on computer supported cooperative work, CSCW ’11. Association for Computing Machinery, New York, pp 395–404. https://doi.org/10.1145/1958824.1958887

Habayeb M, Murtaza SS, Miranskyy A, Bener AB (2018) On the use of hidden markov model to predict the time to fix bugs. IEEE Trans Softw Eng 44(12):1224–1244. https://doi.org/10.1109/TSE.2017.2757480

Hamill M, Goseva-Popstojanova K (2017) Analyzing and predicting effort associated with finding and fixing software faults. Inf Softw Technol 87:1–18. https://doi.org/10.1016/j.infsof.2017.01.002. http://www.sciencedirect.com/science/article/pii/S0950584917300290

Hauge O, Ayala C, Conradi R (2010) Adoption of open source software in software-intensive organizations—a systematic literature review. Inf Softw Technol 52(11):1133–1154. https://doi.org/10.1016/j.infsof.2010.05.008

Hensman J, Fusi N, Lawrence ND (2013) Gaussian processes for big data. In: Proceedings of the twenty-ninth conference on uncertainty in artificial intelligence, pp 282–290

Herzig K, Just S, Zeller A (2013) It’s not a bug, it’s a feature: how misclassification impacts bug prediction. In: Proceedings of the 2013 international conference on software engineering, ICSE ’13. IEEE Press, pp 392–401

Hooimeijer P, Weimer W (2007) Modeling bug report quality. In: Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering, ASE ’07. Association for Computing Machinery, New York, pp 34–43. https://doi.org/10.1145/1321631.1321639

Hu H, Zhang H, Xuan J, Sun W (2014) Effective bug triage based on historical bug-fix information. In: 2014 IEEE 25th international symposium on software reliability engineering, pp 122–132. https://doi.org/10.1109/ISSRE.2014.17

Kim S, Whitehead EJ (2006) How long did it take to fix bugs? In: Proceedings of the 2006 international workshop on mining software repositories, MSR ’06. Association for Computing Machinery, New York, pp 173–174. https://doi.org/10.1145/1137983.1138027

Lazar A, Ritchey S, Sharif B (2014) Improving the accuracy of duplicate bug report detection using textual similarity measures. In: Proceedings of the 11th working conference on mining software repositories, MSR 2014, p 308–311. Association for Computing Machinery, New York. https://doi.org/10.1145/2597073.2597088

Lenarduzzi V, Taibi D, Tosi D, Lavazza L, Morasca S (2020) Open source software evaluation, selection, and adoption: a systematic literature review. In: 2020 46th Euromicro conference on software engineering and advanced applications (SEAA), pp 437–444. https://doi.org/10.1109/SEAA51224.2020.00076

Raja U (2013) All complaints are not created equal: text analysis of open source software defect reports. Empir Softw Eng 18(1):117–138. https://doi.org/10.1007/s10664-012-9197-9

Serrano N, Ciordia I (2005) Bugzilla, itracker, and other bug trackers. IEEE Softw 22(2):11–13. https://doi.org/10.1109/MS.2005.32

Shokripour R, Anvik J, Kasirun ZM, Zamani S (2015) A time-based approach to automatic bug report assignment. J Syst Softw 102(C):109–122. https://doi.org/10.1016/j.jss.2014.12.049

Thung F (2016) Automatic prediction of bug fixing effort measured by code churn size. In: Proceedings of the 5th international workshop on software mining, SoftwareMining 2016. Association for Computing Machinery, New York, pp 18–23. https://doi.org/10.1145/2975961.2975964

Tian Y, Lo D, Xia X, Sun C (2015) Automated prediction of bug report priority using multi-factor analysis. Empirical Softw Engg 20(5):1354–1383. https://doi.org/10.1007/s10664-014-9331-y

Vieira R, da Silva A, Rocha L, Gomes JAP (2019) 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, PROMISE'19. ACM, New York, pp 80–89. https://doi.org/10.1145/3345629.3345639. http://doi.acm.org/10.1145/3345629.3345639

Weiss C, Premraj R, Zimmermann T, Zeller A (2007) How long will it take to fix this bug? In: Fourth international workshop on mining software repositories (MSR'07:ICSE workshops 2007), pp 1–1. https://doi.org/10.1109/MSR.2007.13

Wohlin C, Runeson P, Hst M, Ohlsson MC, Regnell B, Wessln A (2012) Experimentation in software engineering. Springer Publishing Company, Incorporated

Zhang H, Gong L, Versteeg S (2013) Predicting bug-fixing time: an empirical study of commercial software projects. In: 2013 35th International conference on software engineering (ICSE), pp 1042–1051. https://doi.org/10.1109/ICSE.2013.6606654

Zhang X, Yao L, Huang C, Sheng QZ, Wang X (2017) Intent recognition in smart living through deep recurrent neural networks. In: Liu D, Xie S, Li Y, Zhao D, El-Alfy ESM (eds) Neural information processing. Springer International Publishing, Cham, pp 748–758

Zhang X, Chen X, Yao L, Ge C, Dong M (2019) Deep neural network hyperparameter optimization with orthogonal array tuning. In: Gedeon T, Wong KW, Lee M (eds) Neural information processing. Springer International Publishing, Cham, pp 287–295

Zimmermann T, Premraj R, Bettenburg N, Just S, Schroter A, Weiss C (2010) What makes a good bug report? IEEE Trans Softw Eng 36(5):618–643. https://doi.org/10.1109/TSE.2010.63

Affiliations

Renan G. Vieira[1] · **César Lincoln C. Mattos**[1] · **Lincoln S. Rocha**[1] · **João Paulo P. Gomes**[1] · **Matheus Paixão**[2]

César Lincoln C. Mattos
cesarlincoln@dc.ufc.br

Lincoln S. Rocha
lincoln@dc.ufc.br

João Paulo P. Gomes
jpaulo@dc.ufc.br

Matheus Paixão
matheus.paixao@uece.br

1 Federal University of Ceará, Av. Humberto Monte, s/n - Pici, Fortaleza, CE, 60440-593, Brazil

2 State University of Ceará, Av. Dr. Silas Munguba, 1700 - Itaperi, Fortaleza, CE, 60714-903, Brazil