Empirical Software Engineering (2026) 31:43
https://doi.org/10.1007/s10664-025-10771-6

Anticipating bugs: Ticket-level bug prediction and temporal proximity effects

Daniele La Prova[1] · **Emanuele Gentili**[1] · **Davide Falessi**[1]

Received: 25 July 2025 / Accepted: 10 November 2025 / Published online: 15 December 2025

Abstract
Software bugs significantly impact project time, budgets, and safety, motivating extensive research in bug prediction. The primary goal of bug prediction is to optimize testing efforts by focusing on software fragments, i.e., classes, methods, commits (i.e., Just-In-Time or JIT), or lines of code, most likely to be buggy. However, these predictions are made only after defects have already been introduced. Thus, the current bug prediction approaches support fixing rather than prevention. Motivated by the principle of "prevention is better than cure," the aim of this paper is to introduce and evaluate Ticket-Level Prediction (TLP), an approach to identify tickets that will introduce bugs once implemented. We analyze TLP at three temporal points, each point represents a ticket lifecycle stage: Open, In Progress, or Closed. We conjecture that: (1) TLP accuracy increases as tickets progress towards the closed stage due to improved feature reliability over time, and (2) the predictive power of features changes across these temporal points. Our TLP approach leverages 72 features belonging to seven different families: code, developer, external temperature, internal temperature, intrinsic, ticket to tickets, and JIT. Our TLP evaluation uses a sliding-window approach, balancing feature selection and three machine-learning bug prediction classifiers on about 10,000 tickets of two Apache open-source projects. Our results show that TLP accuracy increases with proximity, con- firming the expected trade-off between early prediction and accuracy. Regarding the prediction power of feature families, no single feature family dominates across stages; developer-centric signals are most informative early, whereas code and JIT metrics prevail near closure, and temperature-based features provide complementary value throughout. Our findings complement and extend the literature on bug prediction at the class, method, or commit level by showing that defect predic- tion can be effectively moved upstream, offering opportunities for risk-aware ticket triaging and developer assignment before any code is written.

Keywords Defects · Defect prediction · Machine learning for software engineering

Communicated by Steffen Herbold

Extended author information available on the last page of the article

1 Introduction

Software engineering is the discipline that deals with the cost efficiency, high quality and timely delivery of software systems using quantitative and measur- able approaches (Ghezzi et al. 1991). When developing a project using SE principles, it is not uncommon that developers organize requirements in tickets since they are a good way to keep track of the work (Smith 2016).

Since bugs can significantly impact time, budget and safety, much effort has been spent on bug prediction. The main purpose of bug prediction is to minimize the testing effort. This is achieved by focusing testing on specific software artifacts, such as classes, methods, commits, or lines of code predicted to be buggy. Consequently, significant progress has been made in developing prediction models at the class, method, commit, and line levels (Falessi et al. 2022, 2023; Wei et al. 2016; Kamei and Shihab 2016; Li et al. 2024; McIntosh and Kamei 2018; Ozakinci and Tarhan 2018; Song and Minku 2023; Tantithamthavorn et al. 2020, 2019; Zhao et al. 2023). However, these predicted entities already contain bugs.

Change impact analysis is a software engineering research area focusing on assessing the consequences of modifications to software systems so that teams can minimize unintended consequences, optimize their development efforts, and establish maintenance and testing strategies (Anwer et al. 2019; Aung et al. 2020; Bordin and Benitti 2018; Gentili et al. 2024; Lehnert 2011).

Requirements quality refers to the degree to which software requirements are well-defined, unambiguous, complete, consistent, and testable (Berry and Lawrence 1998; Valdez, et al. 2020). High- quality requirements are essential for guiding development teams and ensuring the final product aligns with stakeholders' needs (Wiegers and Beatty 2013). Berry and Lawrence (Berry and Lawrence 1998) emphasize that clear and precise requirements minimize misunderstandings during development, leading to more efficient project execution, and several studies (Ahonen and Savolainen 2010; Boehm and Basili 2007; Elizabeth et al. 2011) report on the disruptive effects that ambiguity, inconsistency, incompleteness or, more generally, "requirements smells" (Gentili and Falessi 2023) can have on software project success (Kamata and Tamai 2007).

The model's prediction accuracy is impacted by time (Box et al. 2015; Brockwell and Davis 2002; Orrell et al. 2001; Taieb et al. 2009), and it is reasonable to assume that the closer we get to the prediction instant, the more information we gain, and the more precise the prediction becomes. So, the concept of temporal proximity in prediction plays a significant role, as prediction accuracy typically declines over longer time horizons due to the error accumulation of long-term predictors (Box et al. 2015; Orrell et al. 2001), especially when applied to intrinsically stochastic processes, like weather forecasting (Brockwell and Davis 2002; Lorenz 1963), financial stock market (Lo et al. 2011), or epidemic modelling (Keeling and Rohani 2008).

With the idea that prevention is better than cure, we aim to propose and evaluate a first approach for ticket-level prediction (TLP); the approach predicts which tickets, when implemented, will lead to bug injection.

We consider three temporal points to characterize the lifecycle of a ticket: created, assigned, and implemented. We investigate how temporal proximity impacts TLP in terms of the accuracy and power of the predictive features. Specifically, we conjecture that: 1) TLP accuracy improves as the ticket moves closer to implementation (i.e., the bug moves closer

to its injection) due to the increasing reliability of predictive features over time, and 2) the power of predictive features changes over time.

As TLP features, we propose and measure 72 features from commit-level and class-level defect prediction, requirements quality, NLP, and the broader software engineering domain.

Our TLP evaluation considers balancing, feature selection, and many machine-learning bug prediction classifiers on about 11,000 tickets related to two open-source projects from the Apache ecosystem. As TLP accuracy metrics, we use Precision, Recall, F1, AUC, Kappa and GMean. As TLP power metrics, we use the info gain ratio and backward search feature selection.

Our results show that TLP accuracy increases with proximity, confirming the expected trade-off between early prediction and accuracy. Regarding the prediction power of feature families, no single feature family dominates across stages; developer-centric signals are most informative early, whereas code and JIT metrics prevail near closure, and temperature-based features provide complementary value throughout.

Our findings complement and extend the literature on bug prediction at the class, method, or commit level by showing that defect prediction can be effectively moved upstream, offering opportunities for risk-aware ticket triaging and developer assignment before any code is written.

Thus, the contribution of this paper is threefold:

1. We propose a novel framework for predicting defect-prone tickets before any code is implemented.
2. We empirically evaluate the framework across key stages of the ticket lifecycle, highlighting the trade-off between early prediction and accuracy.
3. We conduct a comparative analysis of 72 features spanning defect prediction metrics, natural language processing (NLP), and requirements quality.

The remainder of this paper is structured as it follows. Section 2.2.2 describes how this paper positions itself with past works. Section 3 describes the empirical study design. Section 4 reports the results and section 2.2.1 discusses them. Section 2.2.3 discusses the threats to the study validity. Finally, section 2.2.4 concludes the paper and outlines directions for future work.

2 Ticket level prediction

2.1 Approach

Software projects are usually executed by implementing discrete tasks, com- monly called "tickets" (Bertram 2009). These tickets are assigned to developers who, during the coding process, may inadvertently introduce bugs into the software. The cost of fixing these bugs can increase significantly depending on the software development lifecycle (SDLC) stage at which the bug is detected (Boehm 1981). One way to mitigate bug-fixing costs is early detection. However, while cost-effective, early-stage predictions often have low accuracy due to the limited available information. Conversely, predictions made at later stages benefit from more or updated data; this enhances prediction accuracy but reduces the cost-saving

benefits of early detection. Thus, it is essential to balance the timing, also called the proximity point, and accuracy of bug predictions to optimize resource allocation and minimize the overall cost of software development.

Figure 1 shows the stages of eight specific tickets of the project Apache HIVE as examples. A ticket can be in one of these stages:

- Before Ticket Assignment, aka, Open: The ticket is created and not yet assigned. We measure this proximity point as one second before the ticket assignment date, as reported in JIRA.
- Before First Commit, aka, InProgress: The ticket is assigned and no commit has been submitted yet. We measure this proximity point as one second before the first commit, as reported in Git.
- After Last Commit, aka, Closed: The ticket is completely implemented. We measure this proximity point as one second after the last commit, as reported in Git.

Figure 2 shows a detailed view of such stages applied to a specific ticket, i.e., *HIVE-21783* of type New Feature. In this study, a ticket is bug-inducing if a commit implementing the ticket is buggy, i.e., the commit contains a bug. Figure 2 shows that *HIVE-21783* had eight commits and commit *1475050* is buggy. In fact, another ticket of type Bug was opened,

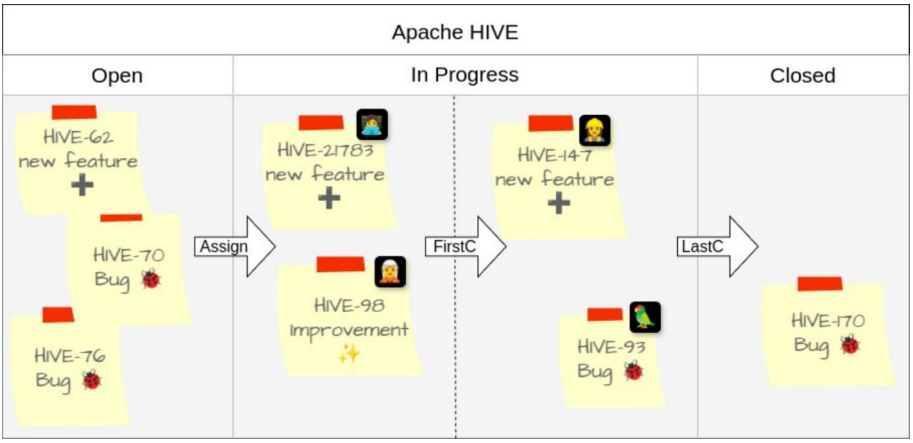


Fig. 1 Example of the lifecycle of Apache HIVE tickets

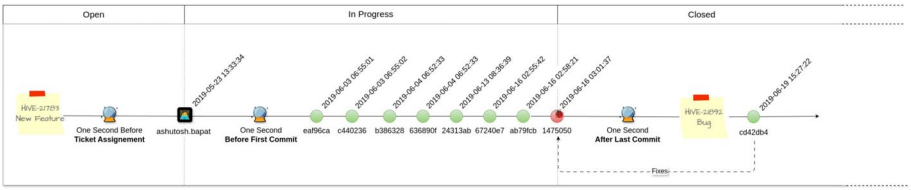


Fig. 2 Example of the proximity points of the HIVE-21783 ticket, its commits and lifecycle, and its fixing process

HIVE-21783, which led to the submission of the commit *cd42db4*. The commit *cd42db4* changed the functionality of some of the lines changed by *1475050*, thus making commit *1475050* buggy and ticket *HIVE-21783* bug-inducing.

We aim to design and evaluate an approach that measures ticket features at different life-cycle stages to predict their likelihood of introducing bugs. Our approach can be formalized as a function mapping ticket features and proximity point to a bug-inducing label, as shown in Equation 1.

$$\text{Input}: \ (\text{Ticket Features}, \ \text{Proximity Point}) \rightarrow \text{Output}: \ \text{isBuggy} \tag{1}$$

where:

Ticket Features are the features we measure for each ticket, as described in the following subsections. The features aim to capture relevant aspects of the development process that can make the ticket more or less bug prone.

- **Proximity Point** is the point in time at which we measure the features, which can be one of the three points described above: Open, InProgress, or Closed.
- **isBuggy** is a boolean value indicating whether the ticket is bug-inducing or not.

We illustrate the approach using the bug-inducing ticket HIVE-170 as an example. We put ourselves in the shoes of a practitioner, such as a Project Manager, aiming at greater control over the project. They take the ticket HIVE-170 and want to hint at how risky it is to implement it, namely, how probable it is that a bug will be injected due to its implementation. Leveraging their knowledge of the project and the problem it solves, they design a set of measurable characteristics deemed indicative of how much the ticket is bug-inducing. These characteristics aim to capture aspects of the ticket that may increase the likelihood of implementation errors, such as:

- The state of the existing codebase;
- The skills of the involved developers;
- How often other tickets have induced bugs in the near past;
- How often the ticket is subject to changes;
- The complexity of the task tracked in the ticket and how clearly it is expressed in natural language;
- The similarity with other bug-inducing tickets
- How it is actually implemented in code.

They then measure these characteristics at the three proximity points described above, obtaining the data shown in Table 1. Unsurprisingly, several features vary in value across the different proximity points. Some features increase over time, like the number of project LOCs, meaning that the project keeps growing in size. Some other features decreased, like the Temporal Locality, meaning that another bug was discovered before or while HIVE-170 was in the Open stage, and no other bugs were discovered after. Besides, other features stay the same, as the Readability Score, suggesting that the ticket description did not change during the ticket implementation. It is worth noting that the more we approach the Closed stage, the higher the measurements reliably represent the features. We can also note that

Table 1 A real-life example illustrating the application of TLP to the buggy ticket HIVE-170 using the RandomForest model

Proximity point	Total LOCs	Assigned developer familiarity	Temporal locality	Activities COUNT
Open	133,168	?	0.051580435	0
InProgress	133,674	0.222221973	0.03363334	15
Closed	133,830	0.222221973	0.017054265	15
Proximity point	Readability score	Max similarity with buggy	Ticket commits entropy	Buggy
Open	34.78003382	0.948700217	?	True
InProgress	34.78003382	0.948700217	?	True
Closed	34.78003382	0.948700217	0.987299689	True

some features are available at specific stages only, as is the case with the Developer Familiarity, which is available only from InProgress stage, namely once a developer has been assigned to the ticket and Ticket Commits Entropy, which is available only from Closed stage, namely once the assigned developer has submitted their commits.

2.2 Feature families

This section presents the features as organized into coherent families, each motivated by prior defect prediction literature across multiple granularities (e.g., class-, method-, line-, and ticket-level) (Falessi et al. 2023; Li et al. 2024; Ozakinci and Tarhan 2018). We adopted a snowballing strategy (Wohlin 2014) to expand the initial set of sources and further incorporated findings from requirements engineering research to identify text- and ticket- based indicators. The final feature set comprises 72 features, grouped into seven semantically coherent families. This section describes the rationale behind each family and provides a few examples. Moreover, for each family we report a summary table describing for each feature the name, implementation code identifier, bibliographic reference, and measurement timing; note that feature values may change over the lifecycle of the ticket. Appendix 1 provides a complete list and definition of each feature.

2.2.1 Code

These features aim to capture the intrinsic characteristics of the underlying system that influence the ease and risk of implementing a ticket, see Table 2 (Kochhar et al. 2016; Zhang 2009). This includes static quality metrics and structural attributes correlating with maintainability and defect-proneness. As an example of feature from this family, we assess the internal quality of the code by counting the number of rule violations detected by PMD

Table 2 TLP features for code (C) family

Name	CodeName	Reference	Availability
Number of smells	*code_quality-smells_count*	Cairo et al. 2018; Falessi, et al. 2020a)	Open
Number of different languages	*code_size-number_of_languages*	Kochhar et al. 2016)	Open
Number of files	*code_size-number_of_files*	Zhang (2009)	Open
Total LOCs	*code_size-total_LOCs*	Zhang (2009)	Open

(Copeland 2005; Falessi et al. 2017), which serves as a proxy for code smells across categories such as design, documentation, error handling, multithreading, and performance (Fowler 1999).

2.2.2 Developer

These features aim to capture the historical performance and familiarity of the ticket assignee, in line with prior work linking developer experience to code quality, see Table 3 (Matsumoto, et al. 2010; Wang, et al. 2020). For instance, we consider the number of tickets assigned to the developer in the past, which serves as a proxy for their experience and familiarity with the codebase. This feature is based on the assumption that developers with more experience are less likely to introduce defects (Patel et al. 2024).

2.2.3 External Temperature

These features aim to capture the project-wide activity that may interfere with development quality, see Table 4 (Falessi, et al. 2020a; Perry et al. 2001). For instance, we consider the number of open tickets in the project, which is a proxy for the workload and potential distractions developers face. This feature is based on the assumption that a higher number of open tickets may lead to increased pressure and reduced attention to detail, potentially resulting in more defects (Falessi et al. 2023).

2.2.4 Internal temperature

Some file-level bug-predicting approaches assume that recently or frequently changed files are more bug-prone, see Table 5 (D'Ambros et al. 2010). We transfer this concept to the ticket level by measuring the "hotness" of the ticket, namely how frequently the ticket was

Table 3 TLP features for developer (D) family

Name	CodeName	Reference	Availability
Assigned Developer's AN- FIC	*assignee-ANFIC*	Matsumoto, et al. (2010)	Assigned
Assigned Developer's Famil- iarity	*assignee-familiarity*	Wang, et al. (2020)	Assigned

Table 4 TLP features for external temperature (E_T) family

Name	CodeName	Reference	Availability
Temporal Locality	*temporal_locality*	Falessi, et al. (2020a; Gu et al. (2021)	Open
Weighted Tempo- ral Locality	*temporal_locality-weighted*	Falessi, et al. (2020a; Gu et al. (2021)	Open
Number of Commits while in progress	*commits_while_in_progress-count*	Perry et al. (2001)	Assigned
Churn of Commits While In Progress	*commits_while_in_progress-churn*	Perry et al. (2001) Faragó et al. (2015)	Assigned
Latest Commit Churn	*latest_commit-churn*	Faragó et al. 2015)	Assigned
Latest Commit Number of Files	*latest_commit-number_of_files*	Keshavarz and Nagappan 2022)	Assigned

Table 5 TLP features for internal temperature (I_T) family

Name	CodeName	Reference	Availability
Ticket Participants Count	*issue_participants-count*	Pinzger et al. 2008)	Open
Activities Count	*activities-count*	Bacchelli et al. 2010)	Open
Comments Count	*activities-comments_count*	Bacchelli et al. 2010)	Open
Work Items Count	*activities-work_items_count*	Bacchelli et al. 2010)	Open
Histories Count	*activities-histories_count*	Bacchelli et al. 2010)	Open
Sentiment Polarity	*nlp4re_sentiment-IT_POL*	Bacchelli et al. 2010) (Zhang and Liu 2017)	Open
Sentiment Subjectivity	*nlp4re_sentiment-IT_SUB*	Bacchelli et al. 2010) (Zhang and Liu 2017)	Open
Number Of Negative Sentiment	*nlp4re_sentiment-CM_NNS*	Valdez, et al. 2020)	Open
Percentage Of Negative Sentiment	*nlp4re_sentiment-CM_PNS*	Valdez, et al. 2020)	Open
Presence Of One Nega tive Sentiment	*nlp4re_sentiment-CM_ONS*	Valdez, et al. 2020)	Open

subject to activities and by how many different stakeholders. For instance, we consider the number of comments added to the ticket, which serves as a proxy for the level of discussion and collaboration surrounding the ticket. This feature is based on the assumption that tickets with more comments may indicate a higher level of complexity or uncertainty, potentially leading to more defects (Falessi et al. 2023). This family also incorporates sentiment analysis of requirement descriptions and comments, emphasizing sentence-level polarity and subjectivity as potential defect indicators (INCOSE 2023; Zhang and Liu 2017). Building on prior findings linking sentiment to software quality and resolution speed (Bacchelli et al. 2010; Valdez, et al. 2020), we incorporate features such as the occurrence and proportion of negative comments to capture affective signals associated with defect-prone tickets.

2.2.5 Intrinsic

Intuitively, some tickets are inherently more challenging to implement than others. These features aim to capture the intrinsic complexity of the ticket, which is not directly related to the code or the developer but instead to the nature of the ticket itself, see Table 6. For instance, we consider the number of requirements in the ticket, which serves as a proxy for the complexity and scope of the ticket. This feature is based on the assumption that tickets with more requirements may be more complex and therefore more likely to introduce defects (Falessi, et al. 2023; Jiang et al. 2007; Patel et al. 2024; Wilson et al. 1997; Winter, et al. 2023). Other features we took into account are the ticket type (e.g., bug, feature request, task) and the priority assigned to the ticket. These features are based on the assumption that different types of tickets may have different defect rates, and that higher-priority tickets may be more likely to introduce defects due to increased pressure and reduced attention to detail (Falessi, et al. 2023; Patel et al. 2024).

2.2.6 Ticket-to-tickets similarity (T2T)

These features aim to capture the semantic similarity to previously bug-inducing, see Table 7. The intuition is that tickets semantically similar to previously bug-inducing ones are more

Table 6 TLP features for Intrinsic (I) family

Name	CodeName	Reference	Availability
Priority	*priority*		Open
Component Count	*components-count*	Patel et al. 2024)	Open
Components Max Bug- giness	*components-max_bugginess*	Patel et al. 2024)	Open
Type	*type*	Gu et al. 2010; Winter, et al. 2023)	Open
Description Attribute Actions	*nlp4re_description-DA_ACT*	INCOSE 2023)	Open
Description Attribute Conditionals	*nlp4re_description-DA_CND*	Jiang et al. 2007; Wilson et al. 1997)	Open
Description Attribute Continuances	*nlp4re_description-DA_CNT*	Jiang et al. 2007; Wilson et al. 1997)	Open
Description Attribute Imperatives	*nlp4re_description-DA_IMP*	Jiang et al. 2007; Wilson et al. 1997)	Open
Description Attribute incompletes	*nlp4re_description-DA_INC*	Jiang et al. 2007; Wilson et al. 1997)	Open
Description Attribute Options	*nlp4re_description-DA_OPT*	INCOSE 2023; Valdez, et al. 2020)	Open
Description Attribute Sources	*nlp4re_description-DA_SRC*	INCOSE 2023)	Open
Description Attribute Weak Phrases	*nlp4re_description-DA_WKP*	Jiang et al. 2007; Wilson et al. 1997)	Open
Description Attribute Risk Level	*nlp4re_description-DA_RKL*	INCOSE 2023; Jiang et al. 2007; Wilson et al. 1997)	Open
Number Of words	*nlp4re_description-EX_CNS*	INCOSE 2023)	Open
Number Of Verbs	*nlp4re_description-EX_VRB*	Wilson et al. 1997)	Open
Number Of Ambiguities	*nlp4re_description-EX_AMG*	INCOSE 2023)	Open
Number Of Directives	*nlp4re_description-EX_DIR*	INCOSE 2023)	Open
Readability Score	*nlp4re_description-EX_RDS*	Wilson et al. 1997)	Open
Sentences completeness	*nlp4re_description-EX_ICP*	Valdez, et al. 2020)	Open
Action Density	*nlp4re_description-EX_ACD*	INCOSE 2023)	Open
Number Of Entities	*nlp4re_description-EX_ENT*	INCOSE 2023)	Open

likely to introduce defects. Similarity is computed using three established NLP techniques: cosine similarity on TF-IDF vectors (Salton and Buckley 1988), Jaccard similarity on token sets (Manning et al. 2008), and Euclidean distance on term frequency vectors (Mikolov, et al. 2013). Each metric is applied to both the ticket title and description, and aggregated via maximum and average operators, leading to 12 distinct features, e.g., 3 techniques * 2 text fields * 2 aggregators.

2.2.7 JIT

These features are the Just-In-Time defect prediction (JIT) features described by Kamei et al. (James, et al. 2023) and Keshavarz and Nagappan (Kamei and Shihab 2016), see Table 8. We neglected the feature "Year" since it is already available as a feature Author date. Since a Ticket can be linked to several commits, we aggregated data using feature-specific strategies (e.g., mean, max, or sum), according to the empirical behavior of the metric.

Table 7 TLP features for Ticket to Ticket (T2T) family

Name	CodeName	Reference	Availability
Max Jaccard Title	*buggy_similarity-max_similarity_jaccard_title*	Jaccard 1901)	Open
Max Jaccard Text	*buggy_similarity-max_similarity_jaccard_text*	Jaccard 1901)	Open
Max TFIDF Title	*buggy_similarity- max_similarity_tfidf_cosine_title*	Baeza-Yates and Ribeiro-Neto 1999	Open
Max TFIDF Text	*buggy_similarity- max_similarity_tfidf_cosine_text*	Baeza-Yates and Ribeiro-Neto 1999	Open
Max Euclidean Title	*buggy_similarity- max_similarity_euclidean_distance_title*	Baeza-Yates and Ribeiro-Neto 1999	Open
Max Euclidean Text	*buggy_similarity- max_similarity_euclidean_distance_text*	Baeza-Yates and Ribeiro-Neto 1999	Open
Average Jaccard Title	*buggy_similarity- avg_similarity_jaccard_title*	Jaccard 1901)	Open
Average Jaccard Text	*buggy_similarity- avg_similarity_jaccard_text*	Jaccard 1901)	Open
Average TF-IDF Title	*buggy_similarity- avg_similarity_tfidf_cosine_title*	Baeza-Yates and Ribeiro-Neto 1999	Open
Average TF-IDF Text	*buggy_similarity- avg_similarity_tfidf_cosine_text*	Baeza-Yates and Ribeiro-Neto 1999	Open
Average Euclidean Title	*buggy_similarity- avg_similarity_euclidean_distance_title*	Baeza-Yates and Ribeiro-Neto 1999	Open
Average Euclidean Text	*buggy_similarity- avg_similarity_euclidean_distance_text*	Baeza-Yates and Ribeiro-Neto 1999	Open

3 Evaluation design

3.1 RQ1: Does temporal proximity impact the accuracy of TLP?

3.1.1 Introduction

In this RQ, we are interested in exploring the tradeoffs between the temporal proximity of the prediction and its accuracy. Specifically, we conjecture that TLP accuracy increases as tickets progress towards the closed stage due to improved feature reliability.

3.1.2 Independent variables

The independent variable of this RQ is the temporal proximity of TLP. This variable has three treatments, i.e., proximity points, as reported in Section 2.1.

Table 8 JIT features

Name	CodeName	References	Availability	Aggregation
Authors Count	*jit-ndev-MAX*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	MAX
Developer Recent Experience	*jit-arexp-MIN*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	MIN
Developer Experience	*jit-aexp-MIN*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	MIN
Developer Subsystem Experience	*jit-asexp-MIN*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	MIN
Modified Subsystems Coun	*jit-ns-MAX*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	MAX
Age	*jit-age-MIN*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	MIN
Author Date	*jit-author_date-DURATION*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	MAX (date)-MIN(date)
LOCs Added	*jit-la-SUM*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	SUM
LOCs Deleted	*jit-ld-SUM*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	SUM
Type	*jit-fix- COUNT_TRUE*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	COUNT(True)
Modified Directories Count	*jit-nd-MAX*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	MAX
Unique Changes Count	*jit-nuc-MAX*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	MAX
Entropy	*jit-ent-MAX*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	MAX
Modified files count	*jit-nf-MAX*	Kamei et al. 2013; Keshavarz and Nagappan 2022)	Closed	MAX
Number of Com- mits	*num_commits*	Falessi et al. 2022	Closed	COUNT

3.1.3 Dependent variables

The main dependent variable of this RQ is the accuracy of TLP. As accuracy indicators of TLP, we used the following six metrics, which are standards in machine learning (Patel et al. 2024):

16 AUC: aka, Area Under the Receiving Operating Characteristic Curve (Falessi et al. 2023), is the area under the curve of true positive rate versus false positive rate, which is defined by setting multiple thresholds. A positive instance is a bug- inducing ticket, whereas a negative instance is a non-bug-inducing ticket. AUC has the advantage of being threshold independent and, therefore, it is recommended for evaluating defect prediction techniques when, like in our TLP context, the costs associated with misclassification errors are unknown or not easily comparable (Lessmann, et al. 2008);

- Precision: It is an accuracy metric describing how often the positive predic- tions of the model are correct (Falessi et al. 2020a);
- Recall: It describes how often the model predicts true positive entities as actually positive (Falessi et al. 2020a).
- Kappa: It describes how well the predictor performs compared against a random guesser (Falessi et al. 2022);
- Specificity: It is a concept close to Recall but related to the negative class. It describes how often the model predicts negative entities as actually negative (Patel et al. 2024);
- GMean: When used to evaluate models performing binary classification tasks, it is defined as the square root of the product between recall and specificity (Patel et al. 2024).

3.1.4 Experimental setup

To reduce variability in the experimental methodology, we took inspiration from what Patel, Adams, and Hassan (Patel et al. 2024) did in JIT context.

Specifically, as the validation technique (Falessi et al. 2020b), we used the same sliding window by Patel, Adams, and Hassan (Patel et al. 2024). We initialize the window for each dataset, taking a batch of the first 1000 instances and split it into 80% training and 20% testing. After each iteration, we update the window by removing the first 200 instances and adding the next 200 in the dataset. At each iteration, the classifier is retrained from scratch on each window's training split and evaluated on that window's test split, preserving temporal order and neglecting data other than the current windows..

Regarding supervised machine learning models, we used the same three used by Patel, Adams, and Hassan (Patel et al. 2024):

- Random Forest (RF): This model is an ensemble learning method that consults a random subset of decision trees whenever it predicts to reduce correlation among the bagged trees (James, et al. 2023);
- Logistic Regression (LR): It is a variant of the linear regression model specialized in binary classification tasks (James, et al. 2023);
- Neural Network (NN): It is a model inspired by the human brain that can learn complex patterns in the data by fitting the weights of the connections between its neurons, which are organized in layers and exchange information through activation functions (James, et al. 2023);

Table 9 reports the hyperparameters used for each model which are default in the used Weka framework (Witten and Frank 2002).

Feature selection uses a correlation-based subset evaluator with greedy search to identify non-redundant predictive subsets (i.e., Weka CfsSubsetEval + GreedyStepwise, backward elimination) (Witten and Frank 2002). Regarding balancing, we evalu- ated without balanc-

Table 9 Weka model parameters

Model	Parameters
Random Forest	-b x -c xx
Logistic Regression	-b x -c xx
Neural Networks	-b x -c xx

ing and with synthetic minority oversampling (i.e., Weka SMOTE with default settings) (Chawla et al. 2002).

3.1.5 Hypothesis and testing

Our null hypothesis H01 is that the accuracy of TLP does not vary across temporal proximity points.

To assess the statistical significance of differences between the treatments, we employed the Friedman test, a non-parametric test suitable for paired data (Friedman 1937). This test ranks the data for each subject across conditions, thereby mitigating the influence of non-normal distributions. The Friedman test efficiently handles the within-subject correlations since each sliding window provides measurements for each of the three proximity points. We tested each accuracy metric for significant differences across treatments within the same dataset and classifier. We set alpha to 0.05 (Gibbons and Chakraborti 2014).

In the context of the Friedman test, a commonly recommended measure of effect size is Kendall's W (Kendall and Babington Smith 1939). It represents the degree of agreement in the rankings across the conditions and is particularly well-suited for repeated measures designs with more than two conditions. Kendall's W ranges from 0 (no agreement) to 1 (complete agreement). In our experimental context, where each window provides measurements under three different proximity points, Kendall's W offers a robust indicator, complementing the Friedman test's chi-square statistic.

3.2 RQ2: Does temporal proximity impact the power of TLP features?

3.2.1 Introduction

Some TLP features can be more or less available and more or less accurate than others at different proximity points. For instance, JIT features are available only when the ticket is Closed and not available when it is Open. Features related to the assigned developer are available when in progress, and can change in Closed. Moreover, the more features we use, the more data we need to train our models. This problem is known in the Machine Learning landscape as the Curse of Dimensionality (Adolfo Crespo Márquez 2022). For the above reasons, a project manager using TLP could be interested in which features are easier to measure and most informative at a specific proximity point. This RQ explores how the importance of each feature varies with proximity.

3.2.2 Independent variables

The independent variables of this RQ are:

1. the temporal proximity of TLP. This has three treatments as in RQ1: Open, In Progress, and Closed.
2. The features used for TLP, which are the 72 detailed in Sect. 2 and grouped in seven families: Code, Developer, External Temperature, Internal Temperature, Intrinsic, Ticket to Tickets, and JIT.

3.2.3 Dependent variables

The dependent variable is the prediction power of features. We used the Infor- mation Gain Ratio (IGR) to measure the prediction power, as we successfully used in a similar work Falessi et al. (Falessi et al. 2020a). One of the advantages of IGR over correlation-based metrics like Spearman is that it is agnostic to classifiers and prediction accuracy metrics. However, the downside is that it is hard to interpret without a given reference (Falessi et al. 2020a). Finally, it is important to note that IGR evaluates a feature's predictive power independently, without considering interactions with other features. However, when a prediction model uses multi- ple features, as in TLP, a feature with low IGR can be more beneficial than a feature with high IGR if it exhibits lower redundancy with the existing feature set. We computed the IGR for all features and examined their distribution across feature families; this allows us to understand how entire families perform in different proximity points. Afterwards, we ranked each feature based on IGR for each project, identifying the top and bottom 10 features in each proximity point. This approach allows us to observe which features are most or least informative and how their importance varies across projects and proximity points.

3.2.4 Hypotheses and testing

Our null hypothesis, H20, is that the power of TLP features does not vary across feature family, temporal locality, and their interaction. As a statistical test, we use the same as RQ1.

3.2.5 Experimental Setup

To produce the evaluation results, we applied the same RQ1 setup. Thus, for each window, we computed the IGR of each feature.

3.3 Measurement procedure

To produce the results, we followed the approach illustrated in Figure 3. First, we needed to find some reusable datasets to train and test the models. Then, we created the TLP datasets to feed the models. Finally, we produced and analyzed the results of prediction accuracy and feature power, answering respectively RQ1 and RQ2.

To create TLP datasets instead of reinventing the wheel and creating a TLP dataset from scratch, we searched for available datasets that could be used or adapted as TLP datasets. Specifically, the TLP dataset requires a ticket to be labelled as bug-inducing or not-bug-

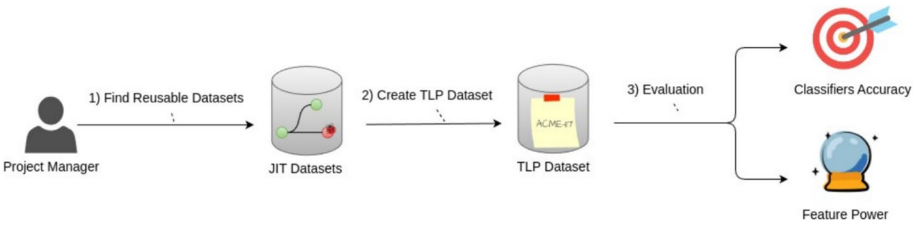


Fig. 3 Measurement procedure

inducing. Since labelling entities as buggy or not is a line of research by itself, we aimed at reusing trustworthy labels. However, we found no dataset labelling tickets as bug-inducing or not- bug-inducing. According to our definition of bug-inducing ticket, we labeled the tickets according to the following rule: *s ticket is bug-inducing if, and only if, at least a commit implementing the ticket is buggy*.

Therefore, we continued our search by looking at JIT datasets providing the buggy/not-buggy labels for commits, and we found three datasets related to three publications (Cabral et al. 2023; Falessi, et al. 2023; Keshavarz and Nagappan 2022). To select the projects in these datasets, we wanted to pick the best projects according to the linkage proportion of buggy tickets; thus, we needed to measure buggy tickets.

To create the TLP dataset, we followed the steps illustrated in Figure 4. Starting from the JIT datasets, we downloaded the tickets linked to the commits following the steps described in algorithm algorithm 1, thus obtaining a set of tickets along with their linked tickets. We labeled the tickets as bug-inducing according to a JIT dataset if they had at least one linked commit labeled as buggy. It is worth noting that different JIT dataset could consider the same commit as buggy or not, depending on the criteria used to label the commits. This issue has been resolved by treating the same project from different datasets as different projects.

To further clarify, let us consider the following example. The commit *eaf96ca* is present in the ApacheJIT dataset as a clean commit. According to the dataset, it has been harvested from the *Apache/hive*. We check the repository logs to measure the commit timestamp and to extract the issue key from the commit message using regular expressions. We found a correspondence with the key HIVE-21783. We then download the ticket details from the JIRA

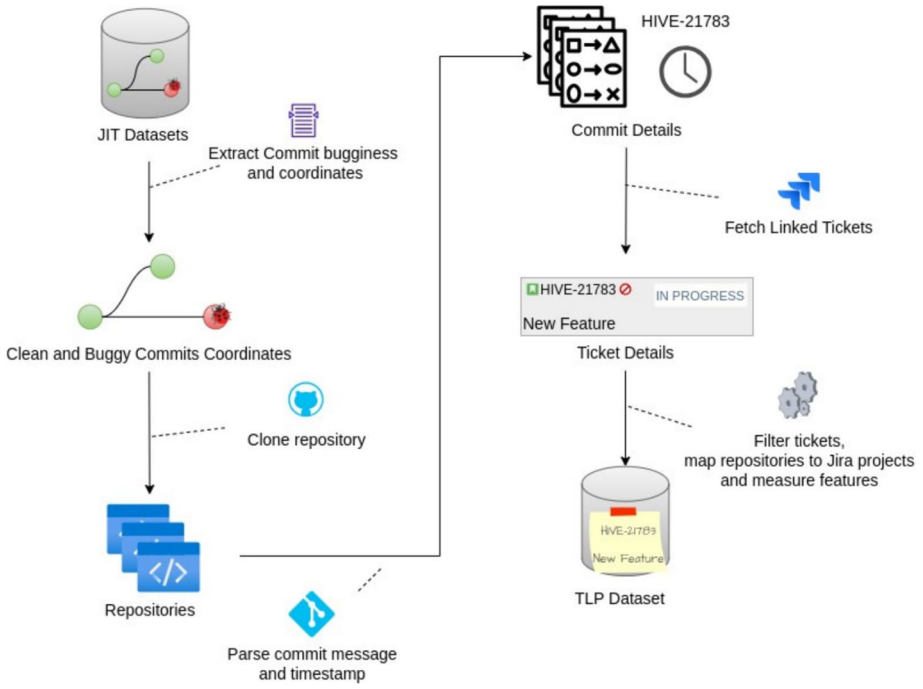


Fig. 4 TLP dataset creation overview

Rest API. Finally, we link the commit to the ticket and save them in our database before moving on to the next commit in the dataset.

Algorithm 1: Set of JIT datasets → Set of linked tickets

- for each JIT Dataset:
  - for each commit reported in the JIT Dataset:
    1. load the commit with its coordinates and JIT features;
       - Commit coordinates include the commit hash and repository.
    2. Get the commit message from the corresponding repository log;
       - If the repository is not available, it is cloned from the corresponding remote host;
       - The projects considered in this study are all versioned using Git and hosted on GitHub.
    3. Scan the commit message for ticket key mentions;
       - An ticket key is a unique identifier for an ticket in a project management system.
       - The format of the key is specific to the project management system.
       - All projects considered in this study use Jira as their project management system.
    4. For each mentioned ticket key:
       (a) Search for matching issues in the project management system;
       (b) For each matching ticket found:
           i. Download ticket details, which include fields and changelog;
              - ticket fields include, but are not limited to: title, description, resolution, due date, opening date, watchers, comments, worklog, votes, attachments, linked issues, subtasks, status, priority, type, assignee, reporter, creator, project, components;
              - ticket changelog consists in a time ordered list of changes to the ticket fields, along with the date and the author of the changes.
           ii. Link ticket to commit;
    5. Load commit timestamp from the corresponding repository.
    6. Store commit along with the mentioned issues.

Note that data in JIRA is known to be imperfect (Herzig et al. 2013; Liu, et al. 2021; Vandehei et al. 2021); To address the problem, we applied a set of filtering steps to the entire set of tickets among all datasets and projects. First, we kept only the tickets which we were able to measure the measurement date in all three proximity points (e.g., If a ticket has no assigned developer in its history, we cannot extract the date of the first assignment). Then, we kept only the tickets belonging to a project with a clearly defined git repository, thus excluding 7291 tickets. We then applied the following set of filters:

- AssignementBeforeFirstCommitDate: We remove a ticket if the date of its first assignment is before its first commit date, since it would not be clear when the ticket transitioned to the In Progress state. A total of 5295 were affected.
- ExclusiveBuggyCommitsOnly: we remove a ticket if all its buggy commits are related to other tickets. This is an important filter since, in this case, we cannot establish which ticket, of the ones related to the buggy commit, induced the bug. A total of 2853 resulted affected.
- CommitAfterOpeningDate: We removed a ticket if the date of its first commit is before its Opening Date. A total of 209 tickets resulted affected.

– NoSnoring: In a past study, we show that removing data from recent releases significantly improves the classifiers' performance (Falessi et al. 2022). Therefore, to avoid the impact of snoring on our dataset, we remove a ticket if it is within the last 20% of the tickets. Note that this was not required for the LeveragingJIT dataset since it was already applied while building the original dataset (Falessi et al. 2023). A total of 9927 ticket resulted affected.

Please note that while just one triggered filter is enough to remove a ticket, each ticket is checked against all filters. Since a ticket could trigger more than a filter at once, the number of affected ticket by each filter may be greater to the total number of removed tickets. Another small print is that we used the first commit date as the measurement date to count the number of affected tickets. When we measured the features, we used three different proximity points. Since some filter depends on the measurement date, the number of affected tickets in this context may be greater.

Table 10 reports the details of available projects. We concluded the search for projects by selecting the projects HIVE and HBASE from ApacheJIT since they have the highest buggy linkage and have lots of usable tickets. Afterwards, we created our TLP dataset by measuring the features described in section 2, producing a dataset for each project and temporal proximity point combination, resulting in six datasets. The tickets in each dataset are

Table 10 List of projects considered in the study, ranked by the percentage of buggy commits linked to Jira tickets. Each project is identified by its name and the datasets its commits come from. The usable ticket count is the number of tickets that have survived the filtering process

Dataset	Project	%Buggy Linkage	%Linkage	#Tickets	#Usable Tickets
leveragingjit	ZooKeeper	100	60	91	91
apachejit	Hive	99	99	6443	5025
apachejit	HBase	97	94	7128	5402
leveragingjit	Tika	96	93	126	124
apachejit	Spark	94	86	1298	631
apachejit	ZooKeeper	93	92	748	559
apachejit	Kafka	83	61	1340	600
apachejit	Camel	83	62	7716	6039
apachejit	Cassandra	81	63	4078	3163
apachejit	Flink	78	48	4265	3295
apachejit	ActiveMQ Classic	74	53	2195	1654
apachejit	Ignite	73	49	2870	2276
apachejit	Zeppelin	73	61	866	291
jitsdp	Camel	67	55	9095	7103
leveragingjit	ActiveMQ Artemis	61	30	133	103
leveragingjit	Qpid	59	47	478	394
apachejit	Groovy	57	39	2612	2004
leveragingjit	Directory ApacheDS	50	15	44	44
leveragingjit	Maven	32	18	255	240
leveragingjit	Nutch	29	24	44	44
leveragingjit	OpenJPA	21	17	69	66
leveragingjit	Groovy	20	17	116	114
apachejit	Hadoop Map/Reduce	17	39	833	661
apachejit	Hadoop HDFS	7	20	2842	2236
apachejit	Hadoop Common	0	99	3249	2586

ordered by the date of the specific proximity point; this is to avoid during prediction, future information is used to predict the past (Falessi et al. 2020b).

4 Evaluation results

In the absence of established guidelines for setting up experiments in TLP, it is first necessary to determine the experimental configuration that yields the highest accuracy before addressing the research questions. Our preliminary analysis, presented in Section A1 of the Appendix, indicates that feature selection and data balancing do not improve the AUC performance in TLP. Consequently, we answer our research questions based on experiments conducted without applying feature selection or balancing techniques. We evaluate the three classifiers independently, no ensemble or model stacking is used, and we report per classifier results. When we show a result without indicating a classifier, then we refer to a simple unweighted average over the three independently trained classifiers, provided solely for trend synthesis.

4.1 RQ1: Does temporal proximity impact the accuracy of TLP?

Figure 5 shows the distributions of TLP accuracy across the three proximity points and a random approach. Table 11 synthesizes Figure 5 by reporting, for each proximity point, the average improvement in TLP accuracy, averaged across our three classifiers, relative to the random baseline.

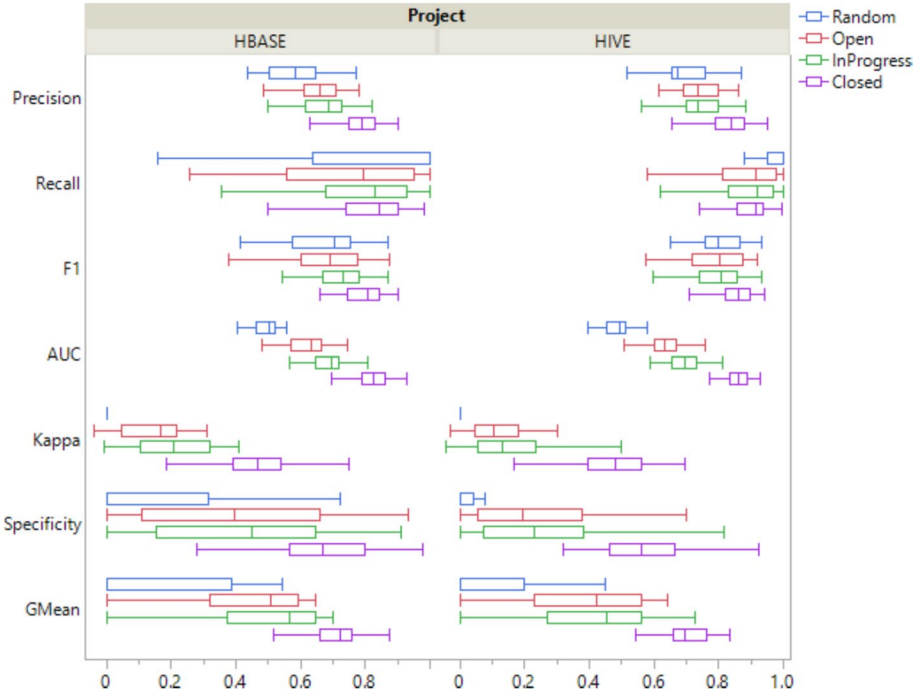


Fig. 5 Distributions of TLP accuracy in three proximity points

Table 11 Average gain across classifiers in TLP accuracy using SW in HBASE and HIVE compared to random TLP

	HBASE			HIVE		
Proximity point	Open	InProgress	Closed	Open	InProgress	Closed
Precision	20%	23%	44%	5%	6%	19%
Recall	−7%	−4%	0%	−10%	−8%	−7%
F1	7%	10%	22%	−3%	−1%	6%
AUC	25%	37%	67%	30%	43%	76%
Kappa*	14%	20%	46%	12%	15%	47%
Specificity	98%	117%	228%	538%	589%	1377%
GMean	163%	193%	314%	311%	364%	663%

Table 12 reports the statistical test results comparing the accuracy of the same classifier on the same project over different proximity points. Table 12 shows statistically significant differences in all accuracy metrics except Recall, indicating that proximity influences TLP accuracy.

4.2 RQ2: Does temporal proximity impact the power of TLP features?

Figure 6 shows, for each feature family, the distribution of TLP accuracy, in terms of AUC, across sliding windows, by proximity point and project.

Figure 7 shows the distribution of IGR across sliding windows for each feature family, grouped by proximity point and project.

Table 13 reports the Statistical test comparison on the impact on IGR of Feature Family, Proximity and their interaction. According to Table 13, we can reject H20 in both projects, and hence we can claim that the prediction power of features varies according to feature family, proximity point, and their interaction.

Finally, Table 14 reports the top 10 features out of 72, ranked by IGR in a specific project in a specific proximity point. Similarly, Table 15 reports the bottom 10 features out of 72, ranked by IGR in a specific project in a specific proximity point.

Regarding Table 15, we note that the analysis has excluded many features since they are not measurable in a specific proximity point. For instance, all the JIT features do not appear in Open and InProgress despite having IGR as 0 by construction. Therefore, Table 15 report the bottom features among the measurable ones

Table 12 Statistical test comparison on the impact on IGR of feature family, proximity and their interaction using sliding-window

	HBASE		HIVE	
Statistical test	Pvalue	KendallsW	Pvalue	KendallsW
Precision	0,0000	0,6775	0,0000	0,6842
Recall	0,0863	0,1225	0,6146	0,0256
F1	0,0000	0,6925	0,0028	0,3102
AUC	0,0000	0,8125	0,0000	0,9058
Kappa	0,0000	0,7500	0,0000	0,7749
Specificity	0,0000	0,6100	0,0001	0,5014
GMean	0,0000	0,7525	0,0000	0,6011

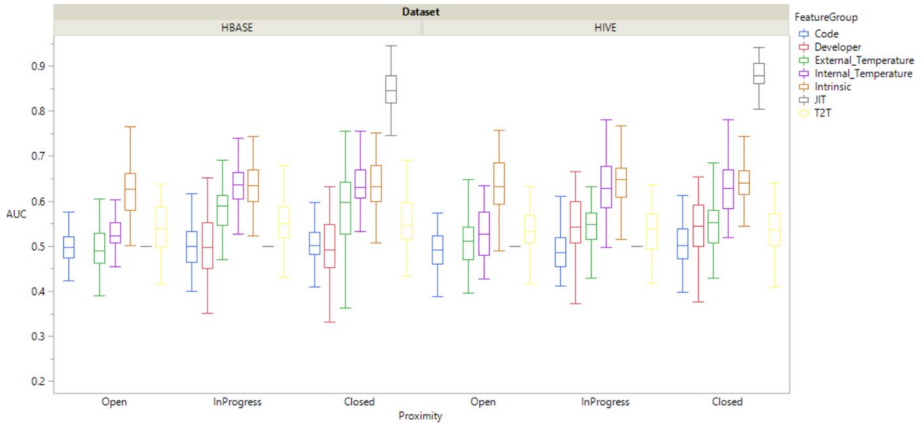


Fig. 6 Distributions of TLP accuracy in terms of AUC achieved by a single feature family, in different proximity points and projects, across windows

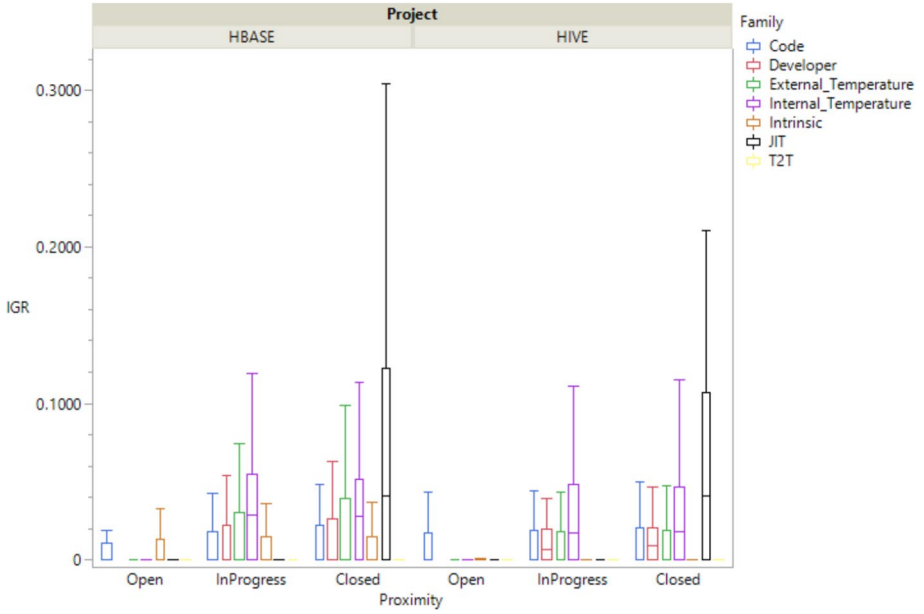


Fig. 7 Distributions of feature family power, in different proximity points and projects, in terms of max and mean IGR, and max and mean selection, across windows

Table 13 Statistical test comparison on the impact on IGR of Feature Family, Proximity and their interaction using moving-window

	HBASE	HIVE
Independent variable	Pvalue	Pvalue
FeatureFamily	0.0001	0.0001
Proximity	0.0001	0.0001
Proximity × FeatureFamily	0.0001	0.0001

Table 14 Top 10 features by IGR per project, grouped by proximity point

Proximity point-open

HBASE				HIVE			
Feature family	Feature name	Mean (IGR)	Rank	Feature family	Feature name	Mean (IGR)	Rank
I	*type*	0,042090	19	I	*type*	0,049482	14
C	*code_size-number_of_languages*	0,021765	33	I	*priority*	0,017441	30
I	*priority*	0,017758	41	C	*code_size-number_of_languages*	0,014348	38
C	*code_quality-smells_count*	0,017296	44	I	*components-max_bugginess*	0,013375	41
C	*code_size-total_LOCs*	0,016226	45	C	*code_size-total_LOCs*	0,009943	45
E_T	*temporal_locality*	0,015672	46	C	*code_size-number_of_files*	0,008898	48
E_T	*temporal_locality-weighted*	0,015567	47	I	*nlp4re_description-DA_ACT*	0,007655	54
I	*nlp4re_description-EX_RDS*	0,014735	51	I_T	*issue_part-count*	0,007394	57
C	*code_size-number_of_files*	0,011611	60	I	*nlp4re_description-EX_VRB*	0,006787	59
I	*nlp4re_description-DA_ACT*	0,010101	65	E_T	*temporal_locality-weighted*	0,006778	60

Proximity point-inprogress

HBASE				HIVE			
Feature family	Feature name	Mean (IGR)	Rank	Feature family	Feature name	Mean (IGR)	Rank
I_T	*activities-count*	0,063784	7	I_T	*activities-histories*	0,060274	7
I_T	*activities-comments*	0,062051	8	I_T	*nlp4re_sentiments-CM_NNS*	0,058154	8
I_T	*activities-histories*	0,049484	15	I_T	*activities-count*	0,053487	11
I_T	*nlp4re_sentiments_CM_NNS*	0,042883	17	I	*type*	0,051830	12
I_T	*issue_participants-count*	0,042722	18	I_T	*activities-comments*	0,043017	15
I	*type*	0,040167	20	I_T	*issue_part-count*	0,032542	20
E_T	*commits_while_in_progress-count*	0,036187	25	E_T	*commits_while_in_progress-count*	0,024242	23
E_T	*commits_while_in_progress-churn*	0,035565	26	I	*priority*	0,019878	28
I_T	*nlp4re_sentiments-CM_PNS*	0,031066	28	E_T	*commits_while_in_progress-churn*	0,016556	32
I	*priority*	0,024077	29	I	*components-max-bugginess*	0,015509	34

Table 14 (continued)

Proximity point-closed							
HBASE				HIVE			
Feature family	Feature name	Mean (IGR)	Rank	Feature family	Feature name	Mean (IGR)	Rank
JIT	*jit-la-SUM*	0,233,243	1	JIT	*jit-la-SUM*	0,157,454	1
JIT	*jit-nf-MAX*	0,70,639	2	JIT	*jit-nf-MAX*	0,142,956	2
JIT	*jit-ent-MAX*	0,148,063	3	JIT	*jit-nd-MAX*	0,137,040	3
JIT	*jit-ld-SUM*	0,128,697	4	JIT	*jit-ent-MAX*	0,113,708	4
JIT	*jit-nd-MAX*	0,126,211	5	JIT	*jit-ld-SUM*	0,111,820	5
JIT	*jit-ns-MAX*	0,068924	6	JIT	*jit-ns-MAX*	0,071429	6
I_T	*activities-count*	0,060702	9	I_T	*activities-histories*	0,056970	9
I_T	*activities-comments*	0,058418	10	I_T	*nlp4re_sentiments-CM_NNS*	0,055771	10
E_T	*commits_while_in_progress-churn*	0,057017	11	I_T	*activities-count*	0,051782	13
JIT	*Jit-ndev-MAX*	0,054313	12	I	*type*	0,042701	16

Table 15 Bottom 10 features by IGR per project, grouped by proximity point

Proximity point-open

HBASE				HIVE			
Feature family	Feature name	Mean (IGR)	Rank	Feature family	Feature name	Mean (IGR)	Rank
I	*nlp4re_description-DA_WKP*	0,000579	159	I	*nlp4re_description-EX_ICP*	0,000354	149
T2T	*buggy_similarity-avg_similarity_tfidt_cosine_title*	0,000438	161	I_T	*activities-comments*	0,000260	151
T2T	*buggy_similarity-avg_similarity_euclidean_distance_title*	0,000366	163	I	*nlp4re_description-DA_CNT*	0,000142	153
I	*components-max_bugginess*	0,000343	164	I	*nlp4re_description-DA_INC*	0,000043	157
I_T	*activities-work_items_count*	0,000000	169	I_T	*activities-work_items_count*	0,000000	159
I_T	*nlp4re_sentiment-IT_SUB*	0,015672	46	C	*nlp4re_description-CM_PNS*	0,000000	160
I	*nlp4re_description-DA_CNT*	0,015567	47	I	*nlp4re_description-DA_SRC*	0,000000	161
I	*nlp4re_description-DA_INC*	0,014735	51	I_T	*nlp4re_description-EX_ENT*	0,000000	162
I	*nlp4re_description-DA_SRC*	0,011611	60	I	*nlp4re_description-EX_RDS*	0,000000	163
T2T	*buggy_similarity-max_similarity_tfidt_cousine_text*	0,010101	65	E_T	*buggy_similarity-max_similarity_jaccard_text*	0,000000	165

Proximity point-inprogress

HBASE				HIVE			
Feature family	Feature name	Mean (IGR)	Rank	Feature family	Feature name	Mean (IGR)	Rank
T2T	*buggy_similarity-max_similarity_tfidt_cousine_title*	0,001537	147	I	*nlp4re_description-DA_INC*	0,000043	155
T2T	*buggy_similarity-max_similarity_tfidt_cousine_text*	0,000915	152	E_T	*latest_commit-churn*	0,000000	166
E_T	*latest_commit-churn*	0,000596	158	I_T	*activities-work_items_count*	0,000000	167
I_T	*nlp4re_sentiment-IT_SUB*	0,000556	160	I	*nlp4re_description-DA_CNT*	0,000000	168
I	*nlp4re_description-DA_WKP*	0,000191	166	I	*nlp4re_description-DA_SRC*	0,000000	169
I_T	*activities-work_items_count*	0,000000	176	I	*nlp4re_description-DA_WKP*	0,000000	170
I	*nlp4re_description-DA_CNT*	0,000000	177	I	*nlp4re_description-EX_AMG*	0,000000	171
I	*nlp4re_description-DA_INC*	0,000000	178	I	*nlp4re_description-EX_ENT*	0,000000	172
I	*nlp4re_description-DA_SRC*	0,000000	179	I	*nlp4re_description-EX_RDS*	0,000000	173
T2T	*buggy_similarity-max_similarity_euclidean_distance_title*	0,000000	181	T2T	*buggy_similarity-max_similarity_jaccard_text*	0,000000	175

Table 15 (continued)

Proximity point-closed							
HBASE				HIVE			
Feature family	Feature name	Mean (IGR)	Rank	Feature family	Feature name	Mean (IGR)	Rank
E_T	*latest_commit-churn*	0,000852	153	I_T	*activities-work_items_count*	0,000000	177
I	*nlp4re_description-DA_WKP*	0,000645	156	I	*nlp4re_description-DA_CNT*	0,000000	178
I_T	*nlp4re_sentiment-IT_SUB*	0,000617	157	I	*nlp4re_description-DA_SRC*	00,000000	179
JIT	*num_commits*	0,000402	162	I	*nlp4re_description-DA_WKP*	0,000000	180
T2T	*buggy_similarity-max_similarity_euclidean_distance_title*	0,000313	165	I	*nlp4re_description-EX_AMG*	0,000000	181
I_T	*activities-work_items_count*	0,000000	182	I	*nlp4re_description-EX_ENT*	0,000000	182
I	*nlp4re_description-DA_CNT*	0,000000	183	I	*nlp4re_description-EX_RDS*	0,000000	184
I	*nlp4re_description-DA_INC*	0,000000	184	JIT	*jit-author_date-DURATION*	0,000000	185
I	*nlp4re_description-DA_SRC*	0,000000	185	JIT	*num-commits*	0,000000	186
JIT	*jit-author_date-DURATION*	0,000000	186	T2T	*buggy_similarity-max_similarity_jaccard_text*	0,000000	16

5 Discussion

5.1 Does temporal proximity impact the accuracy of TLP?

It is intuitive that shifting the proximity point earlier, i.e., moving it to the left, decreases the accuracy of TLP, although such early predictions remain valuable in practice due to their earlier availability. We focus to observe how much the accuracy improves as proximity to the code completion increases, and whether the TLP performance at the earliest proximity point, i.e., in Open, still surpasses that of a random baseline. First of all, according to the statistical tests reported in Table 12 and by the increasing gains over the random baseline with closer proximity points, as shown in Table 11, TLP accuracy improves with proximity with statistically significant differences observed across all seven metrics except Recall.

Focusing the improvement of TLP over a random baseline at various prox- imity points (see Table 11), we observe that the gain varies across different accuracy metrics. For example, in both HBASE and HIVE, there is no ob- servable gain in Recall, not even in the Closed setting. A possible explanation is the highly imbalanced nature of the datasets, which are skewed toward the positive class, making random classification likely to predict positives (Keshavarz and Nagappan 2022). This explanation is supported by the observed gain on Random in Specificity, which is about 200% in the Open proximity point. Since Specificity is analo- gous to Recall but pertains to the negative class, its increase highlights the difficulty of correctly identifying negative instances in imbalanced datasets. Moreover, The smaller gain in Recall could be due to the fact that Recall is insensitive to changes in class prevalence, whereas both Precision and AUC are prevalence-dependent (He and Garcia 2009; Saito and Rehmsmeier 2015).

Comparing gains over the random baseline across different proximity points, we observe that the difference between Closed and InProgress is larger than the difference between InProgress and Open, across all seven metrics and in both projects. This suggests that from a TLP prediction accuracy perspective, the InProgress proximity point is much closer to Open than to Closed, rather than being equidistant or closer to Closed. This finding is somewhat surprising, as we initially expected InProgress to resemble Closed more closely, given that in Open very few features are measurable. One possible explanation is that in Closed, we can leverage JIT features, thus using data about the actual code changes possibly including the bug we aim to predict.

Finally, we note that even in the Open setting, TLP performs substantially better than the random baseline across all seven metrics except Recall. Ac- cording to Table 11, the average gain in Open is 25% for AUC, and Specificity shows an even more remarkable improvement of 535% in HIVE, with notable gains in HBASE also. This suggests that TLP seems useful even before a ticket is assigned to a developer, and that the selected features are able to capture meaningful information about bug-inducing tickets even before most attributes become measurable. his promising finding suggests that TLP can yield meaningful insights even at the earliest ticket stages, potentially enabling bug prevention.

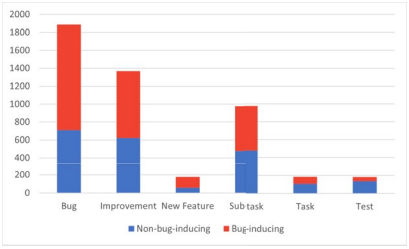
5.2 Does temporal proximity impact the power of TLP features?

According to Fig.s 7 and 5, we observe that the predictive power of features varies significantly based on feature family, proximity point, and their interaction. Specifically, some

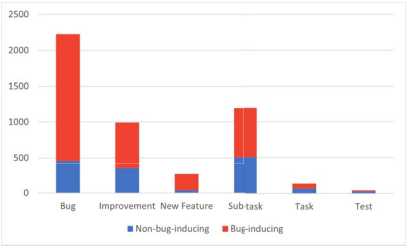
feature families show higher predictive power than others. For instance, the Intrinsic family shows higher predictive power than the Code family across all proximity points. However, for some families, the predictive power varies with proximity. Notably, the JIT family show (by construction) no predictive power at proximity points other than Closed, yet extremely significant at Closed. Therefore, no single feature family consistently outperforms others across all proximity points. Instead, effective prediction models should dynamically adjust feature selection strategies according to the proximity point. This finding emphasises the need to leverage JIT-related features for predictions at later lifecycle stages, while maintaining a broader set of features to ensure robust predictive performance at earlier stages.

Regarding the Top 10 features in the Open setting (Table 15), we observe the following:

- The three most important features in both HBASE and HIVE are *type*, *priority*, and *code_size-number_of_languages*. Notably, *type* exhibits an Information Gain Ratio (IGR) approximately twice as high as that of *priority*, indicating its greater predictive value.
- With respect to *type* (Fig. 8), the absolute number of bugs is highest for tickets labeled as "bugs." However, in terms of proportion, "new features" show the highest ratio of bug-inducing tickets, suggesting that these are the most risk-prone ticket types. Conversely, "test" tickets show the lowest proportion of bug-inducing cases.
- For *priority* (Fig. 9), the category associated with the most bugs in absolute numbers is "Major," while "Blocker" and "Critical" show the highest bug-inducing proportions.

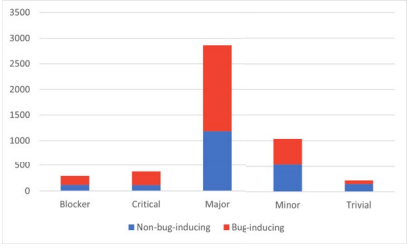


(a) HBASE

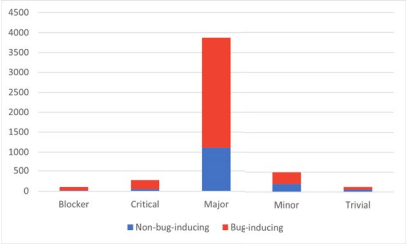


(b) HIVE

Fig. 8 Bug-inducing issue distribution by type



(a) HBASE



(b) HIVE

Fig. 9 Bug-inducing issue distribution by priority

We can observe a general trend: priority appears to positively correlate with proportion of bug-inducing tickets; i.e., the higher the priority the higher the proportion of bug-inducing ticket. However, it is unclear if this is due to a causation or correlation effect. A causal explanation may involve time pressure, i.e., developers may rush to complete high- priority tickets, increasing the chance of introducing bugs. Alternatively, a correlation effect may be that since high-priority tickets are more visible or widely used once implemented, they are more likely to detect and report bugs.

– Three features from the Code family also appear in the top ranks, with *code_size-total_LOCs* being the second most important feature for both HBASE and HIVE. This suggests that the size of the codebase at the time a new ticket is opened may serve as a meaningful proxy for the likelihood of future buggy implementations.

Regarding the Top 10 features in the InProgress and Closed settings (Table 14), we observe a strong prevalence of Internal Temperature features in InProgress and JIT features in Closed. This aligns with expectations, as JIT features are only measurable in Closed. The prominence of Internal Tempera- ture features in InProgress suggests their usefulness for TLP even before ticket closure, likely due to their ability to reflect contextual information and the potential impact of the ticket on the codebase. Furthermore, as expected, the overall ranking of features is higher in Closed than in InProgress, indicating that features are more informative in the Closed setting. This is likely because Closed allows access to JIT features, which include data about the actual code changes—potentially including the bug-inducing modifications themselves. No- tably, the top six features in the Closed stage are consistent across both projects. This is an important finding given that the JIT family consists of 15 features, and that features from other families—such as *activities-count*—exhibit higher IGR scores than some JIT features. This consistency suggests that these top- ranked features are particularly valuable for TLP and may serve as reliable predictors in a JIT-based approach even if not of the JIT family.

Regarding the features with the lowest predictive power, Table 15 shows that the JIT features *num_commits* and *author_date-duration* rank among the lowest, even in the Closed setting where JIT features are fully available. This suggests the need for feature selection within the JIT family and highlights the potential value of incorporating TLP-specific features to enhance prediction in JIT-based models.

6 Related work

This section positions our work within six key research areas: requirements quality, developer-related metrics, change impact analysis, defect prediction, temporal proximity in prediction, and natural language processing (NLP) applied to software tickets.

6.1 Requirements quality

Requirements quality has long been identified as a determinant of downstream software quality. Past works define high-quality requirements as unambiguous, complete, consistent, and testable (Berry and Lawrence 1998; Wiegers and Beatty 2013; Wilson et al. 1997). Deficiencies in these characteristics, i.e., requirements smells (Gentili and Falessi

2023), can propagate into implementation, and impact defects and maintenance costs (Ahonen and Savolainen 2010; Boehm and Basili 2007; Kamata and Tamai 2007). Past works proposed two strategies for mitigating these smells. The first strategy is called preventive, i.e, the promoting high-quality authoring through structured natural language standards (e.g., INCOSE (INCOSE 2023), ISO/IEC/IEEE 29148 ISO/IEC/IEEE (2018). The second strategy is corrective, i.e., using automated smell detectors based on NLP techniques (Femmer et al. 2017; Ferrari, et al. 2018). Both strategies assume the existence of formally specified requirements, a condition rarely met in open-source development, where issue tracking systems (ITSs) like Jira are used (Li 2018). Tickets for open-source projects usually mix functional intent and implementation details with a highly variable structure. Given this lack of standardization, our work diverges from rule-based smell detection and instead models the textual and structural properties of tickets themselves, including features like syntactic completeness, ambiguity, and task granularity. This follows earlier work on more general quality proxies for informal specifications (Carlson and Laplante 2014; Wilson et al. 1997). While these broader features are less actionable than fine-grained smell types, they are more appropriate for predicting defect-inducing changes in non-standardized settings.

6.2 Developer-centric metrics

Developers play a central role in defect introduction. Prior work has shown that factors such as insufficient code ownership, poor responsibility assignment, and variability in developer expertise are strongly associated with fault-proneness (Bergersen et al. 2014; Lee, et al. 2016). In particular, Matsumoto et al. (Matsumoto, et al. 2010) demonstrated the predictive utility of developer-level metrics such as commit frequency and file ownership. Our work extends this line by including both individual and collective developer-related features, such as the number of developers assigned to a module, author diversity in ticket discussions, and historical defect rates of individual developers. These metrics aim to capture socio-technical dimensions of software quality in the context of ticket-level analysis.

6.3 Change impact analysis

Change impact analysis (CIA) aims to predict the effects of modifications in software systems, hence supporting better planning and quality assurance (Anwer et al. 2019; Bordin and Benitti 2018; Lehnert 2011). Past studies have introduced both taxonomies and automated methods for traceability and impact estimation, mainly focusing on predicting code-level changes (Aung et al. 2020; Gentili et al. 2024). In this work we adopt a ticket-centric view of CIA by predicting the effect of implementing a ticket in terms of inducing a bug.

6.4 Defect prediction

Defect prediction remains a central research area in empirical software engi- neering. A large body of work has focused on Just-In-Time Defect Prediction (JITDP), where models assess the likelihood that a given code change will introduce a defect (Kamei and Shihab

2016; Zhao et al. 2023). These models rely heavily on change-level metrics and their performance is influenced by data quality (Li et al. 2024), parameter tuning (Wei et al. 2016), and temporal retraining frequency (Falessi et al. 2022; McIntosh and Kamei 2018). Our approach differs by shifting the prediction granularity from commits to tickets. While prior studies have shown the predictive value of requirements and design artefact quality for early-stage defect prediction (Falessi et al. 2023; Ozakinci and Tarhan 2018), few have formalized this at the level of ITS tickets. We construct a ticket-level dataset derived from manually validated JIT data and integrate static code, developer, and textual features into a unified prediction framework. Our study also explicitly models prediction performance at multiple lifecycle stages (Open, In Progress, Closed), extending the evaluation protocol used in recent benchmark studies (e.g., (Falessi, et al. 2020b)). Our approach primarily differs from previous defect prediction models in granularity and timing. Specifically, previous models focus on predicting the defectiveness of a module (e.g., commit, class, or method) and at code change time, typically leveraging code-centric features (Kamei et al. 2013; Patel et al. 2024) whereas our TLP approach anticipates risk at the ticket-level before the related code exists.

6.5 Temporal proximity in prediction

The timing of prediction affects its accuracy, as widely demonstrated in fields ranging from weather forecasting to financial modelling (Box et al. 2015; Brockwell and Davis 2002; Orrell et al. 2001). Empirical and theoretical studies confirm that predictive performance improves with decreasing temporal distance to the target event due to better information availability and reduced stochastic uncertainty (Graves and Graves 2012; Lorenz 1963; Shumway et al. 2017). Our work is the first to operationalize this principle in the context of TLP. We define and compare model performance across three ticket lifecycle stages (creation, assignment, and closure), thus quantifying the value of temporal proximity in practical software defect prediction. This design choice is methodologically aligned with temporal weighting approaches such as ARIMA and LSTM, prioritizing recent observations in time-series prediction (Florita and Henze 2009; Grover et al. 2015).

6.6 NLP-based analysis of tickets

The widespread use of ITSs such as Jira led to an explosion of unstruc- tured ticket data. To manage this complexity, previous studies applied natural language processing to support ticket classification, prioritization, severity estimation, and duplicate detection (Ahmed et al. 2021; Antoniol, et al. 2018; Choetkiertikul et al. 2019; Falessi et al. 2013; Sun, et al. 2024; Zhang et al. 2016). Building on this foundation, recent work has shown that the textual similarity between new and historical tickets can serve as a proxy for defect risk, particularly when linked to specific code components (Falessi et al. 2020a). Inspired by this, we treat feature request tickets as informal requirements and apply NLP methods to extract syntactic and semantic features. These include both shallow linguistic metrics and semantic similarity measures, which are incorporated into our predictive models. Further details are provided in section 2.

7 Threats to validity

Following established guidelines for empirical research in software engineering (Wohlin et al. 2024), we discuss the threats to validity in four categories: conclusion, internal, construct, and external validity.

7.1 Conclusion validity

Conclusion validity concerns the degree to which conclusions regarding the relationships between the treatment and the outcome are statistically sound and justified (Wohlin et al. 2024). In this study, we used the non-parametric Friedman test (Friedman 1937) to evaluate our hypotheses. Non-parametric tests are less sensitive to assumptions about data distribution and, hence, more appropriate given the nature of defect prediction datasets. While non-parametric tests are typically more conservative and may increase the risk of Type II errors, i.e., failing to reject a false null hypothesis, our results consistently reject the null, suggesting robust differences. We deliberately avoided parametric alternatives such as ANOVA, which assume normality and homoscedasticity and are more susceptible to Type I errors.

7.2 Internal validity

Internal validity refers to the extent to which observed effects can be causally attributed to the treatments rather than confounding variables (Wohlin et al. 2024). A primary threat arises from the absence of ground truth for ticket defectiveness, which could bias the labelling of defect-inducing tickets. To mitigate this, we relied on projects previously vetted in JIT defect prediction studies, e.g., (Kamei et al. 2013; Patel et al. 2024), where established methodologies for linking commits to issue trackers were applied. Furthermore, we did not differentiate among severity levels of the defects, in line with common practice in defect prediction research, e.g., (Falessi, et al. 2023; Tantithamthavorn et al. 2019). While this choice may hide finer-grained distinctions in bug impact, it prevents potential confounding due to inconsistent severity labelling, and it helps preserve the internal consistency of the bug label definition across projects.

7.3 Construct validity

Construct validity addresses the degree to which the operational measures used in the study accurately capture the intended theoretical constructs Wohlin et al. 2024. This threat is particularly relevant in TLP due to the reliance on derived metrics and preprocessed features. To minimize this threat, we based our feature engineering and evaluation design on prior work validated in recent JIT literature (4Patel et al. 2024). One specific concern relates to the decision to avoid hyperparameter tuning of the classifiers. While hyperparameter optimization is known to improve predictive performance (Wei et al. 2016), our goal was not to maximize performance per se, but rather to compare the relative efficacy of TLP across proximity points and feature families. Consequently, using default hyperparameters ensures experimental consistency and avoids confounding due to model overfitting, particularly in a temporally ordered evaluation setting. We opted for text similarity approaches based on TF-IDF, Jaccard similarity, and Euclidean distance of TF vectors, instead of fine-tuned deep

language models like BERT, to maintain interpretability and because our data size could make training large neural models prone to overfitting. Specifically, our dataset of about 10k tickets may be insufficient to fine-tune BERT-based models effectively, risking overfitting. Instead, we relied on well-established, interpretable features (e.g., sentiment, discussion metrics) that have proven effective in defect prediction contexts. Moreover, Guo, Gao, and Jiang (Guo et al. 2024) shows that a simple feed-forward network can perform on par with more sophisticated BERT-based models in JIT; thus, adding a complex model may not outperform our current approach. Anyway, we plan to explore deep learning and LLM models in future work, especially as more data becomes available.

7.4 External validity

External validity concerns how the results can be generalized beyond the studied context (Wohlin et al. 2024). Our study includes two open-source systems with a high commit-to-ticket linkage rate, ensuring high-quality ground truth but limiting the generalizability of the findings to other contexts. The choice to restrict the dataset was intentional: to maintain internal validity and control over data quality. No projects beyond the two selected ones have comparable accuracy in commit-to-ticket linkage; consequently, extending the analysis to additional projects would yield unreliable results. Nonetheless, additional replications across diverse systems—particularly those with different domain characteristics, team structures, or development methodologies—are needed to assess the generalizability of the observed results. Additionally, given the rapid emergence of generative AI for software development (Carleton, et al. 2024; Hou, et al. 2024; Takerngsaksiri, et al. 2025), future research should explore TLP approaches tailored to settings involving automatically generated code, such as those produced by AI-assisted tools. To foster reproducibility and enable external validation, all datasets, scripts, and results are made.[1]

8 Conclusion and future work

Following the principle that prevention is preferable to remediation, this work introduces and evaluates the first approach to Ticket-Level (bug) Prediction, i.e., a method designed to identify tickets likely to introduce bugs upon imple- mentation. The study examines how prediction accuracy and the discriminative power of features evolve with temporal proximity within the ticket lifecycle. Specifically, we investigate three temporal points corresponding to distinct stages in the lifecycle of a ticket: Open, In Progress, and Closed. The central premise underpinning this effort is that even if less accurate, earlier predictions may have higher utility due to their greater potential for enabling proactive quality assurance. Our TLP approach leverages 72 features belonging to seven different families: code, developer, external temperature, internal temperature, intrinsic, ticket to tickets, and JIT.

Our first major finding is that TLP accuracy improves with temporal proximity to ticket closure, with statistically significant differences across most evaluation metrics. However, even at the earliest stage, i.e., Open, when very limited information is available or accurate, TLP models consistently outperform a random baseline in most metrics, especially in AUC and Specificity. This result demonstrates that meaningful predictive info can be extracted

[1] https://doi.org/10.5281/zenodo.15341225

from ticket metadata and static attributes at ticket creation time. This result complements and extends the literature on bug prediction at class, method or commit level (Kamei and Shihab 2016) by showing that defect prediction can be effectively moved upstream, offering opportunities for risk-aware ticket triaging and developer assignment before any code is written. Unexpectedly, we observed that In Progress is closer to Open than Closed regarding predictive accuracy. This finding challenges intuitive assumptions and suggests that the transition from Open to In Progress provides only marginal gains in predictive signal—likely due to the limited availability of new features until the ticket is closed. In contrast, the Closed stage benefits from JIT features that reflect actual code modifications, some of which may be the source of introduced bugs. From a practical standpoint, this implies that the significant leap in prediction capability occurs only upon ticket closure, and that pre-closure models should focus on improving predictive accuracy using only non-code features. A key contribution of this study is showing that no single feature family is uniformly dominant across all stages. Instead, the predictive utility of features is highly contextual: Intrinsic and Code features are most useful early in the lifecycle, Internal Temperature features gain prominence in In Progress, and JIT features dominate in the Closed stage. These findings align with and extend prior work advocating for adaptive, project-sensitive defect prediction models (Tantithamthavorn, et al. 2016). They highlight the need for lifecycle-aware prediction systems that adjust their feature sets dynamically as more information becomes available. Moreover, we show that not all JIT features are equally informative; several rank consistently low even when fully available. This underscores the need for feature selection within strong feature families, a point also raised in earlier studies of defect model stability (Ghotra et al. 2015). Our feature-level analysis further reveals that ticket type and priority are among the most important early-stage predictors, with "new feature" tickets showing the highest proportion of bug-inducing outcomes. Higher-priority tickets (e.g., "Blocker", "Critical") also exhibit elevated bug rates, possibly due to time pressure or heightened visibility. These findings support and refine prior observations regarding the relationship between process factors and defect risk (Herzig et al. 2013; Zimmermann, et al. 2009). From a practical perspective, these attributes offer immediate utility in early-stage triaging, allowing teams to prioritize review and testing resources based on readily available ticket descriptors. An additional noteworthy observation is the cross-project consistency of top-performing JIT features. Despite differences in project characteristics, the top-ranked JIT features at the Closed stage are nearly identical for both HBASE and HIVE, suggesting the existence of robust, transferable signals within this family. However, this consistency contrasts with the observed variability among low-ranking JIT features, indicating that future JIT models may benefit from ticket features.

The TLP scores (i.e., RQ1) can be leveraged at ticket creation or analysis time to flag high-risk tickets for senior assignment and additional QA activities, like additional testing efforts. Such risk-aware triage supports prevention over cure (Kamei et al. 2013; Patel et al. 2024). Moreover, practitioners could leverage the relations in the data used by the prediction model (i.e., RQ2) to understand what aspects of their development process are likely bug-inducing and hence can be improved.

Regarding future work, this study opens several avenues for advancing Ticket-Level Bug Prediction along methodological and practical dimensions. First, we aim to improve the external validity of TLP by extending our anal- ysis to a broader set of open-source and industrial projects. This will enable investigation into how project characteristics—such as

team structure and development process—affect model performance and feature behaviour. Sec- ond, we will pursue refinements in feature engineering, exploring new families such as developer activity metrics and repository evolution indicators. We also plan to leverage deep learning techniques, including domain-specific lan- guage models, to extract semantically rich features from textual and code artefacts. Third, we will evaluate ensemble methods and hybrid architectures that combine traditional machine learning with deep models to enhance predic- tive performance. Real-time prediction capabilities will be explored to enable integration with continuous development pipelines. Fourth, we will investigate the impact of concept drift on TLP stability and develop adaptive learning strategies to maintain accuracy over time, including dynamic retraining and sliding window approaches. Finally, we envision applying TLP in practice by designing developer-facing tools and embedding predictions into issue tracking and CI systems. We also plan to explore its use in software education, providing real-time feedback to students on risk-prone development patterns.

Appendix 1

Feature families

Code-level features

- **Smell Count** (*code_quality-smells_count*): Total number of PMD rule violations in the codebase at the time of ticket creation.
- **Total LOCs** (*code_size-total_LOCs*): Total number of lines of code.
- **Number of Files** (*code_size-number_of_files*): Total number of source files.
- **Number of Languages** (*code_size-number_of_languages*): Number of programming languages used in the codebase.

Developer-related features

- **ANFIC** (*assignee-ANFIC*): Ratio of bug-inducing tickets to total tickets assigned to the developer (Matsumoto, et al. 2010).
- **Familiarity** (*assignee-familiarity*): Proportion of tickets assigned to the developer relative to the total number in the project.

External temperature

- **Temporal Locality** (*temporal_locality*): Ratio of recent bug-inducing tickets in a fixed temporal window.
- **Weighted Temporal Locality** (*temporal_locality-weighted*): Weighted ratio, giving higher weight to temporally closer bugs.
- **Commits During Progress** (*commits_while_in_progress-count*): Number of commits submitted while the ticket was in progress.

- **Churn During Progress** (*commits_while_in_progress-churn*): Cumulative LOCs added, modified, or deleted during the same period.
- **Latest Commit Churn** (*latest_commit-churn*) and **File Count** (*latest_commit- number_of_files*): Change size and scope of the latest commit preceding the ticket.

Internal temperature

- **Ticket Participants Count** (*issue_participants-count*): The more the developers working on a software module, the higher the chance a defect is injected as a result (Pinzger et al. 2008). We measure the number of participants involved in the ticket implementation, i.e. authors of changelog entries, reporter, assignee, creator.
- **Activities Count** (*activities-count*): Developers tend to discuss problematic software entities more (Bacchelli et al. 2010). We transfer this concept to the ticket level by counting ticket Activities. Activities in a ticket can be comments, work items and histories.
- **Comments Count** (*activities-comments*): Ticket participants use comments to express their opinions, ask for clarifications, provide additional information, etc.
- **Work Items Count** (*activities-work_items_count*): Work items are used to track the time spent by participants on the ticket.
- **Histories Count** (*activities-histories*): Histories are used to track the changes made to the ticket.
- **Sentiment Polarity** (*nlp4re_sentiment-IT_POL*): measures the positiveness (or negativeness) of a sentence. It is a number lying between -1 (extremely negative) and 1(extremely positive) (Zhang and Liu 2017).
- **Sentiment Subjectivity** (*nlp4re_sentiment-IT_SUB*): measures the amount of personal opinion and feelings with respect to factual information contained in a sentence. Typically, the higher the index, the less objective is the language used in a sentence (Zhang and Liu 2017). It is a number lying between [0; 1].
- **Number Of Negative Comments** (*nlp4re_sentiment-CM_NNS*): measures the occurrence of negative comments associated to a requirement
- **Percentage Of Negative Comments** (*nlp4re_sentiment-CM_PNS*): measures the occurrence of negative comments divided by the total number fo comments
- **Presence Of One Negative Comment**(*nlp4re_sentiment-CM_ONS*): measures the presence of at least one negative comment

Intrinsic

- **Priority** (*priority*): We measure the priority of the ticket, namely a level of importance telling what ticket should be implemented first. Prioritizing one ticket over another means allocating more time to the former at the cost of the latter, hence the latter could be subject to a rushed development which could produce a buggy implementation. Besides, the higher the priority, the more urgent the ticket is, the more the stress can burden the assignee, leading to a higher chance of mistakes.
- **Components Count** (*components-count*): This feature is inspired by the Entropy feature described in Patel, Adams, and Hassan (Patel et al. 2024). We count the number of project components the ticket is related to, in order to measure the dispersion of the required changes.

- **Components Max Bugginess**(*components-max_bugginess*): We measure the highest bugginess among the components the ticket is related to. We compute the bugginess of the component by counting the number of bug-inducing ticket historically related to the component divided by the total number of tickets related to the component until the measurement date. The idea is that a historically buggy component could be inherently fragile, hence further changes to it could induce more bugs.
- **Type**(*type*): It has been shown that often bug fixing changes induce more bugs in the code (Gu et al. 2010). We extend this concept by considering the change type of the ticket, i.e: bug, improvement, new feature, subtask, etc. It is worth noting than the activity of finding and fixing bugs, although is the most rewarding when successful, can be very frustrating and stressful (Winter, et al. 2023), leading to a higher chance of mistakes.
- **Description Attribute Action** *(nlp4re_description-DA_ACT)*: measures the occurrence of actions by applying patterns to detect obligations and compound verb phrases (INCOSE 2023). The higher, the more probable is that the current requirement should be split into multiple requirements.
- **Description Attribute Conditionals** *(nlp4re_description-DA_CND)*: measures the occurrence of conditional patterns in a sentence, such as the presence of words like "if," "when," "unless," and "depends on," among others (Jiang et al. 2007; Wilson et al. 1997)
- **Description Attribute Continuances** *(nlp4re_description-DA_CNT)*: measures the occurrence of continuance indicators in a sentence, including phrases like "see below," "as follows," "listed," and "in particular," among others (Jiang et al. 2007; Wilson et al. 1997)
- **Description Attribute Imperatives** *(nlp4re_description-DA_IMP)*: measures the occurrence of imperative expressions in a sentence, such as "shall," "must," and "is required to," among others (Jiang et al. 2007; Wilson et al. 1997)
- **Description Attribute Incompletes** *(nlp4re_description-DA_INC)*: measures the occurrence of incomplete markers, identifying acronyms like "TBD," "TBR," "TBC," and "TODO," which indicate missing or pending content (Jiang et al. 2007; Wilson et al. 1997)
- **Description Attribute Options** *(nlp4re_description-DA_OPT)*: measures the occurrence of the use of optionality markers, including words like "can," "could," "may," and "optionally," which indicate non-mandatory elements (Jiang et al. 2007; Wilson et al. 1997)
- **Description Attribute Sources** *(nlp4re_description-DA_SRC)*: measures the occurrence of references to external resources, like files and websites (INCOSE 2023)
- **Description Attribute Weak Phrases** *(nlp4re_description-DA_WKP)*: measures the occurrence of vague or non-assertive phrases that may weaken the clarity and precision of a sentence, e.g. "adequate", "as a minimum", "be capable of" and so on (Jiang et al. 2007; Wilson et al. 1997)
- **Description Attribute Risk Level** *(nlp4re_description-DA_RKL)*: measures the overall level of risk associated to each requirement by summing up each previous index "DA_<>"
- **Number Of Subjects** *(nlp4re_description-EX_SBJ)*: measures the number of general nouns in sentence, identifying subjects and objects (INCOSE 2023)

- **Number Of Words** *(nlp4re_description-EX_CNS)*: measures the number of words in sentence (Wilson et al. 1997)
- **Number Of Verbs** *(nlp4re_description-EX_VRB)*: measures the occurrence of verbs INCOSE 2023)
- **Number Of Ambiguities** *(nlp4re_description-EX_AMG)*: measures the number of ambiguos words used in the sentences, e.g. "some", "many", "few", "often" and so on INCOSE 2023)
- **Number Of Directives** *(nlp4re_description-EX_DIR)*: measures the use of directives-markers that represent instructions or references, such as "e.g.," "i.e.," "figure," "table," "for example," and "note." (Wilson et al. 1997)
- **Readability Score** *(nlp4re_description-EX_RDS)*: measures how readable a piece of text by applying the "Flesch reading ease" score. The lowest is the score, the more technical is the language used in the sentence. (Wilson et al. 1997)
- **Sentence Completeness** *(nlp4re_description-EX_ICP)*: measures whether a sentence is syntactically complete based on the presence of a nominal subject, a verb, and an object (direct, indirect, or prepositional) (Wilson et al. 1997)
- **Action Density** *(nlp4re_description-EX_ACD)*: measures the number of actions with respect the total number of words (INCOSE 2023)
- **Number Of Entities** *(nlp4re_description-EX_ENT)*: measures the number of recognized named entity (NER) in a sentence (INCOSE 2023)

Ticket-to-ticket similarity (T2T)

- **Max Cosine Similarity—Title (*buggy_similarity-max_similarity_tfidf_cosine_title*)**: Maximum cosine similarity between the TF-IDF vector of the current ticket title and those of past bug-inducing ticket titles.
- **Max Cosine Similarity—Description (*buggy_similarity-max_similarity_tfidf_cosine_text*)**: Maximum cosine similarity between ticket descriptions.
- **Avg Cosine Similarity—Title (*buggy_similarity-avg_similarity_tfidf_cosine_title*)**: Average cosine similarity between the current ticket title and past buggy titles.
- **Avg Cosine Similarity—Description (*buggy_similarity-avg_similarity_tfidf_cosine_text*)**: Average cosine similarity over descriptions.
- **Max Jaccard Similarity—Title (*buggy_similarity-max_similarity_jaccard_title*)**: Maximum Jaccard similarity between token sets of ticket titles.
- **Max Jaccard Similarity—Description (*buggy_similarity-max_similarity_jaccard_text*)**: Maximum Jaccard similarity between token sets of descriptions.
- **Avg Jaccard Similarity—Title (*buggy_similarity-avg_similarity_jaccard_title*)**: Average Jaccard similarity across titles.
- **Avg Jaccard Similarity—Description (*buggy_similarity-avg_similarity_jaccard_text*)**: Average Jaccard similarity across descriptions.
- **Max Euclidean Distance—Title (*buggy_similarity-max_similarity_euclidean_distance_title*)**: Maximum Euclidean distance between term frequency vectors of ticket titles.
- **Max Euclidean Distance—Description (*buggy_similarity-max_similarity_euclidean_distance_text*)**: Maximum distance between descriptions.

- **Avg Euclidean Distance—Title (*buggy_similarity-avg_similarity_euclidean_distance_title*)**: Mean Euclidean distance across all title comparisons.
- **Avg Euclidean Distance—Description (*buggy_similarity-avg_similarity_euclidean_distance_text*)**: Mean distance over all description comparisons.

JIT

- **Authors Count** (*jit-ndev-MAX*): We count the highest number of authors involved in a commit linked to the ticket.
- **Developer Recent Experience** (*jit-arexp-MIN*): We measure the lowest recent experience of the authors involved in the commits linked to the ticket.
- **Developer Experience** (*jit-aexp-MIN*): We measure the lowest experience of the authors involved in the commits linked to the ticket.
- **Developer Subsystem Experience** (*jit-asexp-MIN*): We measure the lowest subsystem experience of the authors involved in the commits linked to the ticket.
- **Modified Subsystems Count** (*jit-ns-MAX*): We count the highest number of subsystems modified by a commit linked to the ticket.
- **Age** (*jit-age-MIN*): We measure the lowest temporal distance between a commit linked to the ticket and the most recent commit preceding it.
- **Author Date** (*jit-author_date-DURATION*): We measure the time span between the earliest and the latest commit linked to the ticket.
- **LOCs Added** (*jit-la-SUM*): We sum the LOCs added by all commits linked to the ticket.
- **LOCs Deleted** (*jit-ld-SUM*): We sum the LOCs deleted by all commits linked to the ticket.
- **Type** (*jit-fix-COUNT_TRUE*): We count how many fixing commits are linked to the ticket. Fixing changes are more prone to introduce bugs (Gu et al. 2010).
- **Modified Directories Count** (*jit-nd-MAX*): We count the highest number of directories modified by a commit linked to the ticket.
- **Unique Changes Count** (*jit-nuc-MAX*): We count the highest number of unique changes made by a commit linked to the ticket.
- **Entropy** (*jit-ent-MAX*): We measure the highest entropy of a commit linked to the ticket.
- **Modified Files Count** (*jit-nf-MAX*): We count the highest number of files modified by a commit linked to the ticket.
- **Number of Commits** (*num_commits*): we count the number of commits linked to the ticket to distinguish the cases when commits with different values for every JIT feature produce same values when aggregated.

Setups comparison

Figure 10 reports the average AUC accuracy for the different setups, namely the moving window approach with and without feature selection and with and without SMOTE for balancing.

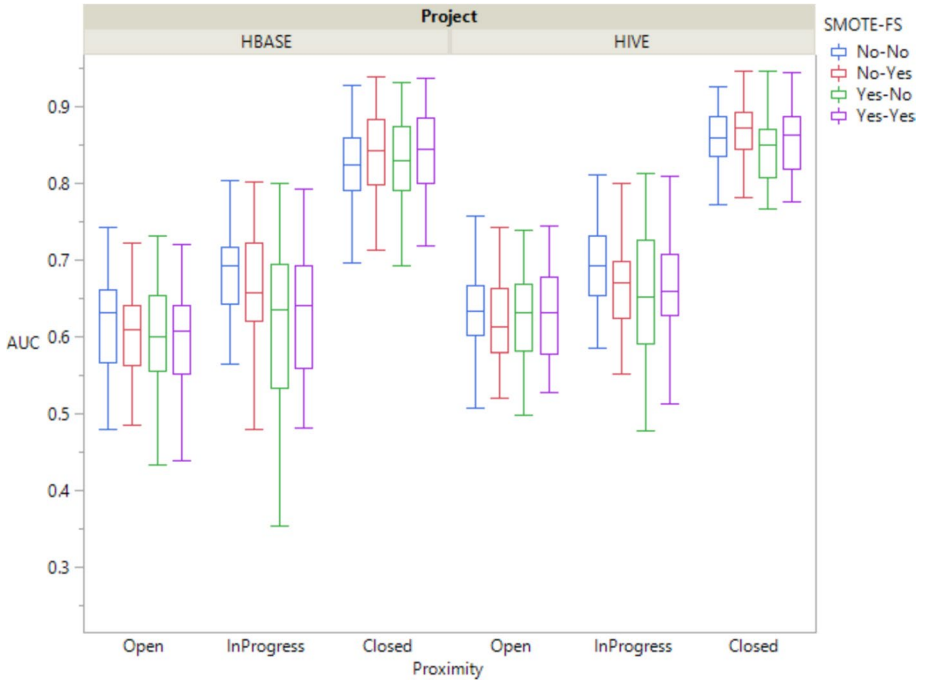


Fig. 10 Average AUC accuracy: moving window approach with and without feature selection and with and without SMOTE balancing

Authors' contributions Conceptualization: DLP, DF Methodology: DLP,DF Software: DLP,EG Validation: DLP,DF Formal analysis: DLP,DF Writ- ing original draft: DLP, DF, EG Writing review editing: EG, DF Supervision: DF Funding acquisition: NA.

Funding Open access funding provided by Università degli Studi di Roma Tor Vergata within the CRUI-CARE Agreement. This research received no external funding.

Data availability The data and code used in this study are available at https://doi.org/10.5281/zenodo.15341225.

Declarations

Competing interests The authors declare that they have no competing interests.

References

Ahmed HA, Bawany NZ, Shamsi JA (2021) CaPBug-a framework for automatic bug categorization and prioritization using NLP and machine learning algorithms. IEEE Access 9:50496–50512. https://doi.org/10.1109/ACCESS.2021.3069248

Ahonen JJ, Savolainen P (2010) Software engineering projects may fail before they are started: post-mortem analysis of five cancelled projects. J Syst Softw 83(11):2175–2187. https://doi.org/10.1016/J.JSS.2010.06.023

Antoniol G et al (2018) Is it a bug or an enhancement?: a text-based approach to classify change requests". In: Proceedings of the 28th annual international confer- ence on computer science and software engineering, CASCON 2018, Markham, Ontario, Canada, October 29–31, 2018. Ed. by Iosif-Viorel Onut et al. ACM, 2018, pp. 2–16. https://dl.acm.org/citation.cfm?id=3291293

Anwer S et al (2019) Comparative analysis of requirement change management challenges between in-house and global software development: findings of literature and industry survey. IEEE Access 7:116585–116611. https://doi.org/10.1109/ACCESS.2019.2936664

Aung TW, Huo H, Sui Y (2020) A literature review of automatic traceability links recovery for software change impact analysis. In: ICPC '20: 28th International Conference on Program Comprehension. Seoul, Republic of Korea, July 13–15, 2020. ACM, pp 14–24. https://doi.org/10.1145/3387904.3389251

Bacchelli A, D'Ambros M, Lanza M (2010) Are popular classes more defect prone? In: Fundamental Approaches to Software Engineering, 13th Inter- national Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20–28, 2010. Proceedings. Ed. by David S. Rosenblum and Gabriele Taentzer. Vol. 6013. Lecture Notes in Computer Science. Springer, pp 59–73. https://doi.org/10.1007/978-3-642-12029-9_5

Baeza-Yates R, Ribeiro-Neto B (1999) Modern information retrieval, vol 463. ACM press New York

Bergersen GR, Sjøberg DI, Dybå T (2014) Construction and validation of an instrument for measuring programming skill. IEEE Trans Softw Eng 40(12):1163–1184. https://doi.org/10.1109/TSE.2014.2348997

Berry DM, Lawrence B (1998) Requirements engineering. IEEE Softw 15(2):26–29

Bertram D (2009) The social nature of issue tracking in software engineering. PhD thesis. University of Calgary. https://www.proquest.com/openview/6b8f9d3b9c6e5f1a3b3e8e3e3e3e3e3e/1?pq-origsite=gscholar&cbl=18750

Boehm BW (1981) Software engineering economics. Prentice Hall, 1981. isbn: 978-0138221225

Boehm B, Basili VR (2007) Software defect reduction top 10 list". In: Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research 34.1. p 75

Bordin AS, Benitti FB (2018) Software mainte- nance: what do we teach and what does the industry practice?" In: Proceedings of the XXXII Brazilian Symposium on Software Engineering, SBES 2018, Sao Carlos, Brazil, September 17–21, 2018. Ed. by Uirá Kulesza. ACM, pp 270–279. https://doi.org/10.1145/3266237.3266251

Box GE et al (2015) Time series analysis: forecasting and control. John Wiley & Sons

Brockwell PJ, Davis RA (eds) (2002) Introduction to time series and forecasting. Springer

Cabral GG et al (2023) An investigation of online and offline learning models for online just-in-time software defect prediction. Empir Software Eng 28(5):121. https://doi.org/10.1007/s10664-023-10335-6

Cairo AS, Carneiro GDF, Monteiro MP (2018) The impact of code smells on software bugs: a systematic literature review. Information 9(11):273. https://doi.org/10.3390/INFO9110273

Carleton A et al (2024) Generative AI: redefining the future of software engineering. IEEE Softw 41(6):34–37. https://doi.org/10.1109/MS.2024.3441889

Carlson N, Laplante P (2014) The NASA automated requirements measurement tool: a reconstruction. Innov Syst Softw Eng 10(2):77–91. https://doi.org/10.1007/S11334-013-0225-8

Chawla NV et al (2002) SMOTE: synthetic minority over-sampling technique. J Artif Intell Res 16:321–357. https://doi.org/10.1613/JAIR.953

Choetkiertikul M et al (2019) A deep learning model for estimating story points. IEEE Trans Softw Eng 45(7):637–656. https://doi.org/10.1109/TSE.2018.2792473

Copeland T (2005) PMD applied (vol 10)

Crespo Márquez A (2022) The curse of dimensionality. In: Digital maintenance management: guiding digital transformation in maintenance. Springer, pp 67–86

D'Ambros M, Lanza M, Robbes R (2010) An extensive comparison of bug prediction approaches. In: 2010 7th IEEE working conference on mining software repositories (MSR 2010). pp 31–41. https://doi.org/10.1109/MSR.2010.5463279

Falessi D et al (2020a) Leveraging historical associations between requirements and source code to identify impacted classes. IEEE Trans Softw Eng 46(4):420–441

Falessi D et al (2020b) On the need of preserving order of data when validating within-project defect classifiers. Empir Softw Eng 25(6):4805–4830. https://doi.org/10.1007/s10664-020-09868-x
Falessi D, Cantone G, Canfora G (2013) Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques. IEEE Trans Softw Eng 39(1):18–44. https://doi.org/10.1109/TSE.2011.122
Falessi D, Russo B, Mullen K (2017) What if i had no smells?" In: Bener A, Turhan B, Biffl S (eds) 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9–10, 2017. IEEE Computer Society, pp 78–84. https://doi.org/10.1109/ESEM.2017.14
Falessi D, Ahluwalia A, Penta MD (2022) The impact of dormant defects on defect prediction: a study of 19 apache projects. ACM Trans Softw Eng Methodol (TOSEM) 31(1):1–26. https://doi.org/10.1145/3467895
Falessi D et al (2023) Enhancing the defectiveness prediction of methods and classes via JIT. Empir Softw Eng 28(2):37. https://doi.org/10.1007/s10664-022-10261-z
Faragó C, Hegedűs P, Ferenc R (2015) Cumulative code churn: Impact on maintainability. In: Godfrey MW, Lo D, Khomh F (eds) 15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27–28, 2015. IEEE Computer Society, pp 141–150. https://doi.org/10.1109/SCAM.2015.7335410
Femmer H et al (2017) Rapid quality assurance with requirements smells. J Syst Softw 123:190–213. https://doi.org/10.1016/j.jss.2016.02.047
Ferrari A et al (2018) Detecting requirements defects with NLP patterns: an industrial experience in the railway domain. Empir Softw Eng 23(6):3684–3733
Florita AR, Henze GP (2009) Comparison of short-term weather forecasting models for model predictive control. HVAC&R Research 15(5):835–853
Fowler M (1999) Refactoring-improving the design of existing code. Addison Wesley object technology series. Addison-Wesley. isbn: 978–0–201–48567–7. http://martinfowler.com/books/refactoring.html
Friedman M (1937) The use of ranks to avoid the assumption of normality implicit in the analysis of variance. In: Journal of the American Statistical Association32.200, pp 675–701. issn: 01621459, 1537274X. http://www.jstor.org/stable/2279372 (visited on 04/10/2025)
Wei Fu, Menzies T, Shen X (2016) Tuning for software analytics: Is it really necessary? Inf Softw Technol 76:135–146. https://doi.org/10.1016/J.INFSOF.2016.04.017
Gentili E, Falessi D (2023) Characterizing requirements smells. In: Kadgien R et al (eds) Product-focused software process improvement - 24th International conference, PROFES 2023, Dornbirn, Austria, December 10–13, 2023, Proceedings, Part I, vol 14483. Lecture Notes in Computer Science. Springer, pp 387–398
Gentili E, Carka J, Falessi D (2024) A systematic mapping study on impact analysis. In: Fill H-G et al (eds) Proceedings of the 19th International Conference on Software Technologies, ICSOFT 2024, Dijon, France, July 8–10, 2024. SCITEPRESS, pp 375–382. https://doi.org/10.5220/0012758200003753
Ghezzi C, Jazayeri M, Mandrioli D (1991) Fundamentals of software engi- neering. Prentice-Hall, Inc
Ghotra B, McIntosh S, Hassan AE (2015) Revisiting the impact of classification techniques on the performance of defect prediction models. In: Bertolino A, Canfora G, Elbaum SG (eds) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, Volume 1. IEEE Computer Society, pp 789–800. https://doi.org/10.1109/ICSE.2015.91
Gibbons JD, Chakraborti S (2014) Nonparametric statistical inference: revised and expanded. CRC Press
Graves A, Graves A (2012) Long short-term memory. Supervised sequence labelling with recurrent neural networks. <book-title update="added">Supervised sequence labelling with recurrent neural networks. pp 37–45
Grover A, Kapoor A, Horvitz E (2015) A deep hybrid model for weather forecasting. In: Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining, pp 379–386
Gu Z et al (2010) Has the bug really been fixed?" In: Kramer J et al (eds) Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1–8 May 2010. ACM, pp 55–64. https://doi.org/10.1145/1806799.1806812
Gu X et al (2021) Do bugs propagate? An Empirical analysis of temporal corre- lations among software bugs. In: Møller A, Sridharan M (eds). https://doi.org/10.4230/LIPICS.ECOOP.2021.11
Guo Y, Gao X, Jiang B (2024) An empirical study on JIT defect prediction based on BERT-style model. In: CoRR abs/2403.11158. https://doi.org/10.48550/ARXIV.2403.11158
He H, Garcia EA (2009) Learning from imbalanced data. IEEE Trans Knowl Data Eng 21(9):1263–1284. https://doi.org/10.1109/TKDE.2008.239

Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Notkin D, Cheng BHC, Pohl K (eds) 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18–26, 2013. IEEE Computer Society, pp 392–401. https://doi.org/10.1109/ICSE.2013.6606585

Hou X et al (2024) Large language models for software engineering: a systematic literature review. ACM Trans Softw Eng Methodol 33(8):1–79

Elizabeth M, Hull C, Jackson K, Dick J (eds) (2011) Requirements engineering, Third edition. Springer. isbn: 978–1–8499–6404–3. https://doi.org/10.1007/978-1-84996-405-0

INCOSE (2023) INCOSE systems engineering handbook. John Wiley & Sons

ISO/IEC/IEEE (2018) International Standard - Systems and software engineering – Life cycle processes – Requirements engineering. In: ISO/IEC/IEEE 29148:2018(E), pp 1–104. https://doi.org/10.1109/IEEESTD.2018.8559686

Jaccard P (1901) Étude comparative de la distribution florale dans une portion des Alpes et des Jura. Bull Soc Vaudoise Sci Nat 37:547–579

James G et al (2023) An introduction to statistical learning: With applications in python. Springer Nature

Jiang Y, Cukic B, Menzies T (2007) Fault prediction using early lifecycle data. In: ISSRE 2007, The 18th IEEE International Symposium on Software Reliability, Trollhättan, Sweden, 5–9 November 2007. IEEE Computer Society, pp 237–246. https://doi.org/10.1109/ISSRE.2007.24

Kamata MI, Tamai T (2007) How does requirements quality relate to project success or failure?" In: 15th IEEE International Requirements Engineering Conference, RE 2007, October 15–19th, 2007, New Delhi, India. IEEE Computer Society, pp 69–78. https://doi.org/10.1109/RE.2007.31

Kamei Y, Shihab E (2016) Defect prediction: Accomplishments and future challenges. In: Leaders of Tomorrow Symposium: Future of Software Engineering, FOSE@SANER 2016, Osaka, Japan, March 14, 2016. IEEE Computer Society, pp 33–45. https://doi.org/10.1109/SANER.2016.56

Kamei Y et al (2013) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng 39(6):757–773. https://doi.org/10.1109/TSE.2012.70

Keeling MJ, Rohani P (2008) Modeling infectious diseases in humans and animals. Princeton University Press

Kendall MG, Smith BB (1939) The problem of m rankings. Ann Math Stat 10(3):275-287. Issn:00034851 http://www.jstor.org/stable/2235668 (visited on 04/10/2025).

Keshavarz H, Nagappan M (2022) ApacheJIT: A large dataset for just-in-time defect prediction. In: 19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23–24, 2022. ACM, pp 191–195. https://doi.org/10.1145/3524842.3527996

Kochhar PS, Wijedasa D, Lo D (2016) A Large scale study of multiple programming languages and code quality. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14–18, 2016 - Volume 1. IEEE Computer Society, pp 563–573. https://doi.org/10.1109/SANER.2016.112

Lee T et al (2016) Developer micro interaction metrics for software defect prediction. IEEE Trans Softw Eng 42(11):1015–1035

Lehnert S (2011) A taxonomy for software change impact analysis. In: Cleve A, Robbes R (eds) Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution, EVOL/IWPSE 2011, Szeged, Hungary, September 5–6, 2011. ACM, pp 41–50. https://doi.org/10.1145/2024445.2024454

Lessmann S et al (2008) Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans Softw Eng 34(4):485–496

Li P (2018) Jira software essentials: plan, track, and release great applications with Jira software. Packt Publishing Ltd

Li Z, Du Q, Zhang H, Jing XY, Wu F (2024) An empirical study of data sampling techniques for just-in-time software defect prediction. Autom Softw Eng 31(2):56. https://doi.org/10.1007/S10515-024-00455-8

Liu S et al (2021) An extensive empirical study of inconsistent labels in multi- version-project defect data sets. In: CoRR abs/2101.11749. arXiv: 2101. 11749. http://arxiv.org/abs/2101.11749

Lo AW, MacKinlay AC (2011) A non-random walk down wall street. Princeton University Press

Lorenz EN (1963) Deterministic nonperiodic flow. J Atmos Sci 20(2):130–141

Manning CD, Raghavan P, Schütze H (2008) Introduction to information retrieval. Cambridge University Press, Cambridge, UK

Matsumoto S et al (2010) An analysis of developer metrics for fault prediction. In: Menzies T, Koru G (eds) Proceedings of the 6th International conference on predictive models in software engineering, PROMISE 2010, Timisoara, Romania, September 12–13, 2010. ACM, p 18. https://doi.org/10.1145/1868328.1868356

McIntosh S, Kamei Y (2018) Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction. IEEE Trans Softw Eng 44(5):412–428. https://doi.org/10.1109/TSE.2017.2693980

Mikolov T et al (2013) Linguistic regularities in continuous space word representations. In: Proceedings of the international conference on learning representations (ICLR) http://arxiv.org/abs/1301.3781

Orrell D et al (2001) Model error in weather forecasting. Nonlinear Process Geophys 8(6):357–371

Ozakinci R, Tarhan A (2018) Early software defect prediction: a systematic map and review. J Syst Softw 144:216–239. https://doi.org/10.1016/J.JSS.2018.06.025

Patel H, Adams B, Hassan AE (2024) Post deployment recycling of machine learning models: don't throw away your old models! Empir Softw Eng 29(4):100. https://doi.org/10.1007/s10664-024-10492-2

Perry DE, Siy HP, Votta LG (2001) Parallel changes in large-scale software development: an observational case study. ACM Trans Softw Eng Methodol (TOSEM) 10(3):308–337. https://doi.org/10.1145/383876.383878

Pinzger M, Nagappan N, Murphy B (2008) Can developer- module networks predict failures? In: Harrold MJ, Murphy GC (eds) Proceedings of the 16th ACM SIGSOFT international symposium on foundations of software engineering, 2008, Atlanta, Georgia, USA, November 9–14, 2008. ACM, pp 2–12. https://doi.org/10.1145/1453101.1453105

Saito T, Rehmsmeier M (2015) The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. PLoS One 10(3):e0118432. https://doi.org/10.1371/journal.pone.0118432

Salton G, Buckley C (1988) Term-weighting approaches in automatic text retrieval. Inf Process Manage 24(5):513–523

Shumway RH (2017) ARIMA models. Time series analysis and its applications: with R examples. pp 75–163

Smith K (2016) Managing electronic resource workflows using ticketing system software. Ser Rev 42(1):59–64

Song L, Minku LL (2023) A procedure to continuously evaluate predictive performance of just-in-time software defect prediction models during software development. IEEE Trans Softw Eng 49(2):646–666

Sun W et al (2024) Method-level test-to-code traceability link construction by semantic correlation learning. IEEE Trans Softw Eng. https://doi.org/10.1109/TSE.2024.3449917

Taieb SB et al (2009) Long-term prediction of time series by combining direct and MIMO strategies. In: International joint conference on neural networks, IJCNN 2009, Atlanta, Georgia, USA, 14–19 June 2009. IEEE Computer Society, pp 3054–3061. https://doi.org/10.1109/IJCNN.2009.5178802

Takerngsaksiri W et al (2025) Human-in-the-loop software development agents. arXiv: 2411.12924 [cs.SE]. https://arxiv.org/abs/2411.12924. Accessed 27 Nov 2025

Tantithamthavorn C, Hassan AE, Matsumoto K (2020) The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans Softw Eng 46(11):1200–1219

Tantithamthavorn C et al (2016) An empirical comparison of model validation techniques for defect prediction models. IEEE Trans Softw Eng 43(1):1–18

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2019) The impact of automated parameter optimization on defect prediction models. IEEE Trans Softw Eng 45(7):683–711. https://doi.org/10.1109/TSE.2018.2794977

Valdez A et al (2020) Sentiment analysis in Jira software repositories. In: 2020 8th International conference in software engineering research and innovation (CONISOFT). pp 254–259. https://doi.org/10.1109/CONISOFT50191.2020.00043

Vandehei B, Costa DAD, Falessi D (2021) Leveraging the defects life cycle to label affected versions and defective classes. ACM Trans Softw Eng Methodol (TOSEM) 30(2):1–35

Wang C, Li Y, Chen L, Huang W, Zhou Y, Xu B (2020) Examining the effects of developer familiarity on bug fixing. J Syst Softw 169:110667

Wiegers KE, Beatty J (2013) Software requirements. Pearson Education

Wilson WM, Rosenberg LH, Hyatt LE (1997) Automated analysis of requirement specifications. In: Richards Adrion W et al (eds) Pulling together, proceedings of the 19th international conference on software engineering, Boston, Massachusetts, USA, May 17–23, 1997. ACM, pp 161–171. https://doi.org/10.1145/253228.253258

Winter E et al (2023) How do developers really feel about bug fixing? Directions for automatic program repair. IEEE Trans Softw Eng 49(4):1823–1841. https://doi.org/10.1109/TSE.2022.3194188

Witten IH, Frank E (2002) Data mining: practical machine learning tools and techniques with Java implementations. ACM Sigmod Rec 31(1):76–77

Wohlin C (2014) Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Shepperd MJ, Hall T, Myrtveit I (eds) 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, May 13–14, 2014. ACM, pp 38:1–38:10. https://doi.org/10.1145/2601248.2601268

Wohlin C et al (2024) Experimentation in software engineering, second edition. Springer. isbn: 978–3–662–69305–6. https://doi.org/10.1007/978-3-662-69306-3

Zhang H (2009) An investigation of the relationships between lines of code and defects. In: 25th IEEE international conference on software maintenance (ICSM 2009), September 20–26, 2009, Edmonton, Alberta, Canada. IEEE Computer Society, pp 274–283. https://doi.org/10.1109/ICSM.2009.5306304

Zhang L, Liu B (2017) Sentiment analysis and opinion mining. In: Sammut C, Webb GI (eds) Encyclopedia of machine learning and data mining. Springer, pp 1152–1161

Zhang T et al (2016) Towards more accurate severity prediction and fixer recom- mendation of software bugs. J Syst Softw 117:166–184. https://doi.org/10.1016/J.JSS.2016.02.034

Zhao Y, Damevski K, Chen H (2023) A systematic survey of just-in-time software defect prediction. ACM Comput Surv 55(10):1–35. https://doi.org/10.1145/3567550

Zimmermann T et al (2009) Cross-project defect prediction: a large scale exper- iment on data vs. domain vs. process. In: van Vliet H, Issarny V (eds) Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amster- dam, The Netherlands, August 24–28, 2009. ACM, pp 91–100https://doi.org/10.1145/1595696.1595713

Authors and Affiliations

Daniele La Prova[1] · Emanuele Gentili[1] · Davide Falessi[1]

✉ Davide Falessi
Falessi@ing.uniroma2.it

Daniele La Prova
daniele.laprova@hotmail.it

Emanuele Gentili
emanuele.gentili@mbda.it

[1] University of Rome Tor Vergata, Rome, Italy