Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Ranking Code Clones to Support Maintenance Activities

Osama Ehsan · Foutse Khomh · Ying Zou · Dong Qiu

Received: date / Accepted: date

Abstract Developers often reuse code fragments by copy-and-paste activities to speed up code delivery. Through this copy-and-paste process, they create duplicated code, also known as code clones. As the software system evolves, the number of clones can increase substantially and impact code quality negatively. Prior studies have shown that inconsistent changes on code clones can introduce bugs in a software system and clones that have experienced some specific evolutionary patterns being more at risk than others. As the number of clone copies increases in a software system. it becomes tedious and time-consuming for developers to track and maintain all code clones. Recent studies have proposed approaches to analyze the clone evolution history for better clone maintenance. However, these approaches do not provide a specified list of code clones at a granular level (i.e., commits) that can help developers prioritize their clone maintenance activities. It is important to track the code clone changes at the commit level, as developers can fix/refactor code clones early.

In this paper, we leverage machine learning to develop clone ranking models that can help developers identify the most risky clones early on. Specifically, we

Osama Ehsan
Department of Electrical and Computer Engineering
Queen's University, Kingston, Ontario, Canada
E-mail: osama.ehsan@queensu.ca

Ying Zou
Department of Electrical and Computer Engineering
Queen's University, Kingston, Ontario, Canada
E-mail: ying.zou@queensu.ca

Foutse Khomh
Polytechnique Montréal, Canada
E-mail: foutse.khomh@polymtl.ca

Dong Qiu
Huawei Technologies Co., Ltd.
E-mail: qiudong624@huawei.com

detect clones from 52 projects (34 Java and 18 C) that have 534,672 commits and build 469,239 clone genealogies. We extract 28 features capturing the characteristics of code clones at commit level. We then train learning-to-rank (LtR), classification, and regression machine learning models to rank the code clones based on fault occurrence during their evolutionary history.

Our comparison of machine learning approaches indicates that classification (for the probability of being faulty) and regression (for the proportion of faulty changes) perform well in ranking code clones. Multiple unique developers who change a code clone and the age of a code clone (in terms of the number of cloned code changes) have a significant effect on the risk of faults in the code clones. Our results can help developers identify the most risky code clones first and prioritize them for refactoring to prevent future faults.

Keywords code clones · clone genealogies · learning-to-rank · regression approaches · mixed-effect models · clone evolutionary patterns

1 Introduction

Similar or identical code fragments in software projects are referred to as **code clones**. A code fragment is a continuous section of source code that has a well-defined *start* and *end* and implements certain functionality. For example, a method implementing the addition of two numbers. A code clone is a code fragment or sub-string of code appearing at least twice in the source code of a program, a pair of clones is called a clone pair. Code clones can be either syntactically or semantically similar [23]. Developers may use existing code fragments by copy-and-paste activities to implement a similar functionality rather than writing the code from scratch. Once a code clone is created, it undergoes multiple changes and evolves over time. The set of states and changes between the states of a clone pair, across the revisions forms a **clone genealogy**.

Over the lifetime of a project, the number of code clones may increase. For example, at the start of project for nd4j[1], there are 109 clones. As the project evolved, the number of code clones increased to 9,550. With such a large amount of code clones in a system, it becomes crucial to maintain them, since (1) multiple, possibly unnecessary duplication of code can increase maintenance cost [14], and (2) inconsistent changes to the cloned code can introduce faults leading to incorrect product behaviours [23].

Existing approaches provide clone detection tools that can identify a large number of code clones. However, it can be tedious and time-consuming for the developers to examine all the code pairs in a project manually to identify the risky clone pairs. For example, there can be multiple code clones in any commit, and it is difficult for a developer to recognize which code clones should be tracked first to prevent future faults.

[1] https://github.com/deeplearning4j/nd4j

Recent studies have used code clone evolution history to analyse the risk of different clone evolution patterns [3], predict the consistency-maintenance of code clones [55], and identify short-lived code clones [46]. Although these previous work can help improve the maintenance of the code clones, they do not support developers in assessing the risk of all the clones in a software system. To the best of our knowledge, no prior studies aimed at ranking the code clones in a software system, at the commit level, based on their fault-proneness.

In this study, we examine the potential of various machine learning techniques at providing an accurate ranking of code clones based on their fault-proneness. We are interested to rank code clones at the commit level. The commit is the smallest unit of code change in a repository, and code clones can be refactored as soon as they are introduced or modified if they are tracked at the commit level[32]. We use 52 projects (i.e., 34 Java and 18 C) with an average of 893k SLOC per project to train the machine learning models. Using the history of changes of a software system, we detect code clones and build clone genealogies. Then, we leverage the SZZ algorithm [41] to identify buggy commits in the clone genealogies. Next, we calculate 28 clone-related metrics on the detected clones and train learning-to-rank (LtR), classification, and regression machine learning models, to rank the code clones for fault-proneness. We also conduct an analysis of the most important features of the models, to understand the main factors affecting the fault-proneness of a code clone. We assess the performance of the models and answer the following research questions:

RQ1: Can we use learning-to-rank (LtR) algorithms to effectively rank fault-prone code clones? In this research question, we train multiple learning-to-rank (LtR) algorithms to assess their effectiveness at ranking clones. We experiment with both early projects and mature projects to allow developers to rank code clones during the early stages of the development process of their projects. Our results show that the best performing approach (i.e., LightGBM) is able to achieve a precision of 0.72 (for Java projects). Learning-to-rank algorithms achieve moderate performance due to the unavailability of the labeled ranked data for code clones.

RQ2: How well can classification algorithms rank fault-prone clones? In this research question, we use classification algorithms to rank clones based on the probability of being faulty (i.e., fault-proneness of code clones). To support clone ranking in early development phases as well as mature development phases, we constructed specialized models for the projects that have limited clone history in the early phase as well as projects that have more longer clone history in the mature phase. We performed 10-fold cross-validation of the trained models and found that Random Forest achieves the highest AUC among the studied classification approaches. Our analysis also shows that the performance of the models increases as more information about the history of code clones is available.

RQ3: Can we use regression algorithms to predict the proportion of faulty changes in code clones and effectively rank fault-prone clones? In this research question, we rank code clones based on their predicted proportion of faulty changes, using regression algorithms. Similar to the previous research question, we also train two specialized models for early stage and mature stage respectively. Our 10-fold cross-validation results show that LightGBM and XGBOOST are able to identify the fault-prone code clones with a high R-squared score (0.89 for Java projects). Similar to RQ2, the performance of the models increase as more information about the history of code clones is available.

RQ4: Which metrics are significantly correlated to the risk of faults in code clones? In this research question, we investigate the most influential features in determining the rank of the code clones. We use a mixed-effect model to determine the most significant features when ranking code clones. Results suggest that developers should closely monitor the code clones that have more changes and the code clones that are changed by multiple developers.

Overall, this paper makes the following key contributions.

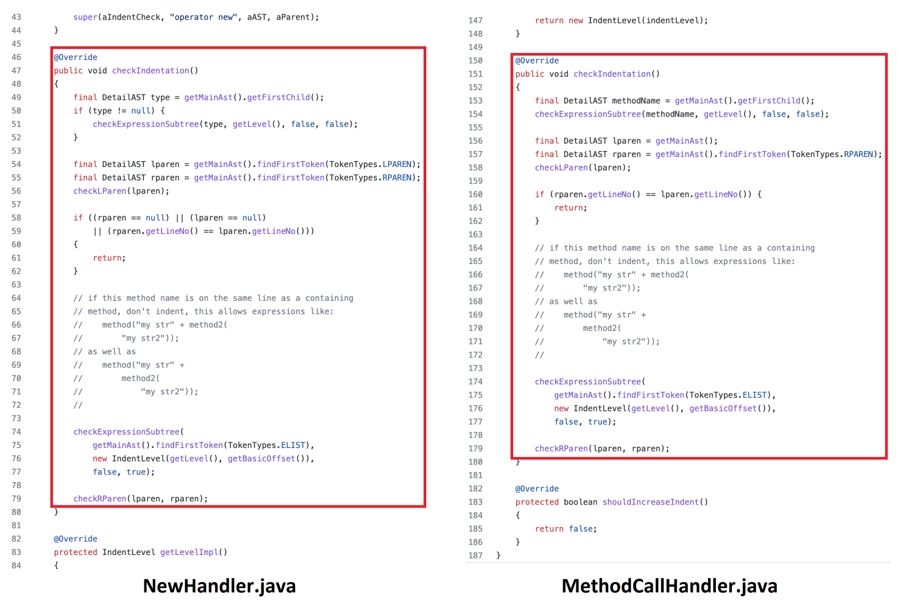
– We present the result of a large-scale empirical study that assesses the efficiency of learning-to-rank, classification, and regression machine learning models at providing an accurate ranking of code clones based on their fault-proneness.
– We provide approaches for ranking code clones and demonstrate the suitability of regression techniques for ranking code clones.
– We provide specialized models for early projects and mature projects to aid software developers at different phases of the software development process.
– We perform a qualitative analysis of the studied projects to identify the most significant features that correlate with fault occurrences in cloned code.

The trained machine learning models presented in this paper can help software maintainers identify the most risky code clones early on to prevent the introduction of faults.

Paper Organization. The rest of the paper is organized as follows. Section 2 provides background information on the multiple machine learning algorithms used in the paper. Section 3 presents the design of our case study, which includes project selection, clone detection, clone genealogy, fault identification using SZZ, and feature calculation. Section 4 presents the results of our four research questions. Section 5 discusses the implications of our results for developers. Section 6 summarizes the prior studies and related work in the code clone domain. Section 7 discusses the threats to the validity of our study. Finally, Section 8 concludes our work and outlines some avenues for future work.

2 Background

This section presents background information on motivating example of ranking code clones and multiple machine learning ranking techniques.



```java
        super(aIndentCheck, "operator new", aAST, aParent);
    }

    @Override
    public void checkIndentation()
    {
        final DetailAST type = getMainAst().getFirstChild();
        if (type != null) {
            checkExpressionSubtree(type, getLevel(), false, false);
        }

        final DetailAST lparen = getMainAst().findFirstToken(TokenTypes.LPAREN);
        final DetailAST rparen = getMainAst().findFirstToken(TokenTypes.RPAREN);
        checkLParen(lparen);

        if ((rparen == null) || (lparen == null)
            || (rparen.getLineNo() == lparen.getLineNo()))
        {
            return;
        }

        // if this method name is on the same line as a containing
        // method, don't indent, this allows expressions like:
        //    method("my str" + method2(
        //        "my str2"));
        // as well as
        //    method("my str" +
        //        method2(
        //            "my str2"));
        //

        checkExpressionSubtree(
            getMainAst().findFirstToken(TokenTypes.ELIST),
            new IndentLevel(getLevel(), getBasicOffset()),
            false, true);

        checkRParen(lparen, rparen);
    }

    @Override
    protected IndentLevel getLevelImpl()
    {
```

```java
        return new IndentLevel(indentLevel);
    }

    @Override
    public void checkIndentation()
    {
        final DetailAST methodName = getMainAst().getFirstChild();
        checkExpressionSubtree(methodName, getLevel(), false, false);

        final DetailAST lparen = getMainAst();
        final DetailAST rparen = getMainAst().findFirstToken(TokenTypes.RPAREN);
        checkLParen(lparen);

        if (rparen.getLineNo() == lparen.getLineNo()) {
            return;
        }

        // if this method name is on the same line as a containing
        // method, don't indent, this allows expressions like:
        //    method("my str" + method2(
        //        "my str2"));
        // as well as
        //    method("my str" +
        //        method2(
        //            "my str2"));
        //

        checkExpressionSubtree(
            getMainAst().findFirstToken(TokenTypes.ELIST),
            new IndentLevel(getLevel(), getBasicOffset()),
            false, true);

        checkRParen(lparen, rparen);
    }

    @Override
    protected boolean shouldIncreaseIndent()
    {
        return false;
    }
}
```

Fig. 1 An example of clone pair. The code inside the red boxes are clone pairs from checkstyle project.

2.1 Motivating Example

We use an example to illustrate the need to rank the buggy commits. In one of the commits in project nd4j[2], there are 87 code clones identified from the clone detection tool (one of the clone pairs is shown in Figure 1). Suppose that there are 48 faulty code clones. Currently, in practice, it is a manual process to check 87 code clones in order to identify the potential faulty code clones and fix them. Development teams are likely to have limited time and resources, so they would need to prioritize which code clones to examine and test first. However, some commits have a higher number of code clones identified by the clone detection tools (e.g., a commit in nd4j has more than 600 code clones identified). In such scenarios, it becomes imperative for the developers to have a ranked list, so they can focus on the most risky code clones first by making optimal use of their resources.

2.2 Learning-to-Rank Algorithms

Learning-to-Rank (LtR) algorithms stem from the application of machine learning models (either supervised or semi-supervised) or reinforcement learning to solve ranking tasks. The significant difference between LtR and classical

[2] https://github.com/eclipse/deeplearning4j/commit/0ec1c8f

regression/classification problems is that LtR focuses on a list of items to predict the rank, while regression/classification problems focus on one item at a time. The most common application of LtR algorithm is web search, while it is also used in product recommendation and candidate selection.

Types of LtR Models. In general, there are three types of LtR algorithms: pointwise, pairwise, and listwise. To understand the working of different types, let us suppose to have the following ranking task. Given a list of documents `D=` `d`$_1$`, d`$_2$`,....d`$_n$, and a query `q`, we would like to learn a function `f` such that `f(q,D)` will predict the relevance of the documents with respect to the query. In our context, the list of clone pairs at a commit is the documents and our query to identify fault-prone clone pairs. The three types of LtR algorithms are different in terms of loss function formulation in the machine learning task.

- **Pointwise.** The documents are scored individually. It is useful in cases where the relevance of the given documents is binary.
- **Pairwise.** The training example is constituted of pairs and are given relevancy scores ranging from lowest to highest. For example, 0 is irrelevant and 5 is the most relevant.
- **Listwise.** The training examples consist of all the documents available to rank. This method is costly; however, it covers the full range of documents that need to be ranked.

The pairwise and listwise algorithms are reported to outperform the pointwise algorithms [29]. Therefore, in this paper, we experiment only with pairwise and listwise algorithms using the following two popular frameworks; i.e., XGBOOST and LightGBM. Specifically, we use the pairwise algorithms (RankBoost, and RankNet), the listwise algorithms (LambdaRank, LambdaMart, and Random Forest), and the frameworks XGBOOST, LightGBM to rank code clones.

Evaluation of LtR Models.

Several metrics have been proposed to evaluate the LtR models. They are generally divided into two categories: binary relevance and graded relevance.

(1) Binary Relevance. Binary relevance evaluation is used when the ranking task only considers the relevance or irrelevance of documents.

(a) Mean Average Precision (MAP). MAP is a measure based on the binary label of relevancy where precision needs to be calculated at every ranking position. The average ranking is calculated at every relevant position while every irrelevant position is penalized. Equation 1 is used to calculate the MAP.

$$MAP = \frac{\sum_{q=1}^{Q} AP(q)}{Q} \tag{1}$$

where

$AP(q) =$ Average precision given at query q

$Q =$ Total number of queries

(2) Graded Relevance. Graded relevance metrics are used to evaluate the LtR algorithms when the ranking task gives different scores to the documents based on their relevancy to the query.
(a) Normalized Discounted Cumulative Gain (NDCG). Discounted cumulative gain prefers the higher relevant item to be ranked higher. In Equation 2, the numerator is an increasing function of relevance while the denominator is a decreasing function of ranking position. Therefore, higher relevance gains more points.

$$DCG@k = \sum_{i=1}^{k} \frac{2^{l_i} - 1}{log_2(i+1)} \tag{2}$$

$$\begin{aligned} &\text{where} \\ DCG &= \text{Discounted cumulative gain} \\ IDCG &= \text{Ideal discounted cumulative gain} \\ l &= \text{relevance score} \end{aligned}$$

Normalized discounted cumulative gain (NDCG) is calculated using Equation 3. If DCG@k is calculated by re-sorting the list by correct relevance labels, then the IDCG@k is the maximum possible value of DCG@k that can be achieved given a ranking list.

$$NDCG@k = \frac{DCG@k}{IDCG@k} \tag{3}$$

$$\begin{aligned} &\text{where} \\ DCG &= \text{Discounted cumulative gain} \\ IDCG &= \text{Ideal discounted cumulative gain} \end{aligned}$$

2.3 Classification Approaches

Classification approaches use selected machine learning algorithms to learn how to assign a class label to examples from a problem domain. The class labels can be binary (i.e., only two classes to chose from, e.g., 0 or 1) or multiple (i.e., more than two classes, e.g., Types of grade for course A, B, C, or F). From a modeling perspective, classification requires a training dataset that has many example inputs (i.e., features) and outputs.

Evaluation of Classification Techniques.
The following four metrics are commonly used to evaluate a classification technique.

Precision is the fraction of relevant instances among the retrieved instances [18]. As shown in Equation 4, we use precision to identify the total number of correctly identify fault-prone code clones (true positives) over the the number of the wrongly identified fault-prone code clones (false positives).

$$P = \frac{TP}{TP + FP} \tag{4}$$

Recall is defined as a probability that a relevant object is returned by a system [18]. As shown in Equation 5, we use recall to identify the number of fault-prone clone pairs that are correctly identified (true positives) over the number of the missed fault-prone clone pairs (false negatives).

$$R = \frac{TP}{TP + FN} \tag{5}$$

F1-score is the weighted average of the precision and the recall that includes the impact of the false positive as well as false negatives. Hence, we use F1-score to evaluate the overall accuracy of our approach for our manually labeled dataset (Sample$_{tuning}$), as shown in Equation 6.

$$F1 = 2\frac{P.R}{P + R} \tag{6}$$

AUC (Area under the ROC Curve). The discriminative power of the model measures the ability of the model to distinguish value 0 and 1 of the dependant variable (i.e., predicting the probability of the code clones being buggy). We use Area Under Curve (AUC) [19] to determine the discriminative power of the model. We use Receiver Operator Curve (ROC) to plot the true positives against the false positive for different thresholds. The value of AUC ranges from 0 to 1, 0 being the worst performance, 0.5 being the random guessing performance, and 1 being the best performance. [19]

2.4 Regression Analysis

Regression analysis is a type of statistical method used to determine the strength of a relationship between a dependent variable (usually denoted as Y) and a number of independent variables. In contrast to only two possible outcomes in the classification problem, a regression problem can have multiple outcomes. For example, predicting the price of a house or the interest rate over a period of time. There are multiple types of regression analysis that include linear regression, logistic regression, ridge regression, and lasso regression.

Evaluation of Regression Approaches

Multiple metrics have been introduced in the literature to evaluate the performance of regression models. Two of them are most commonly used by the practitioners: (1) R-Squared (R^2) and (2) Root mean square error (RMSE)

(1) R-Squared (R^2). R-squared is a proportional improvement in the prediction of a regression model as compared to the mean model. The scale of R^2 ranges from zero to one, with zero indicating that the regression model is not able to improve than the mean model while one depicts that the regression model performs perfectly in terms of prediction. Equation 7 is used to calculate R^2.

$$R^2 = \frac{\text{Error from regression model}}{\text{Simple average model}} \tag{7}$$

(2) Root Mean Square Error (RMSE). The RMSE is the square root of the variance of the residuals that indicates the absolute fit of the model. In simple words, it calculates how close are the predicted values and the actual values. Lower values of RMSE are desirable for regression models. Equation 8 is used to calculate RMSE.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} Predicted_i - actual_i}{N}} \tag{8}$$

2.5 Machine Learning Frameworks

This section briefly describes the machine learning frameworks used in this paper.

XGBOOST. XGBOOST is a scalable machine learning system for tree boosting proposed by Chen and Guestrin [7]. XGBOOST combines the original model (i.e., gradient boosting) with weak base learning models in an iterative manner to generate a robust learning model. The residual in each iteration of the boosting is used to improve the previous predictor (i.e., optimizing the loss function). The algorithm is engineered for time efficiency and memory optimization. One of the most important features of the XGBOOST is the sparse awareness that makes it useful even when some of the data values are missing. XGBOOST offers a block structure that enables the algorithm to make use of parallelization for tree construction. The models in XGBOOST are constructed by computing the gradient descent using an objective function.

XGBOOST has been used in prior studies for cross-device identification [47], intrusion detection [11], and fault localization [53].

LightGBM. LightGBM is a gradient boosting framework that uses tree-based algorithms [26]. The most distinctive feature of LightGBM among other tree-based algorithms is that it grows trees vertically. In other words, LightGBM grows tree leaf-wise while other tree-based algorithms grow trees level-wise. This allows it to reduce more loss than a level-wise algorithm. LightGBM is highly effective for large scale data. LightGBM supports GPU learning, but it is sensitive to the size of datasets; there is a risk of overfitting if used on a relatively smaller dataset. Similar to XGBOOST, LightGBM computes gradient descent using an objective function.

LightGBM has been used in prior studies for intrusion detection [45] and malware detection [34].

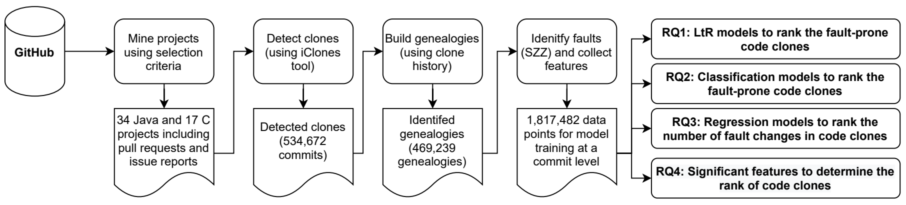

GitHub
Mine projects using selection criteria
34 Java and 17 C projects including pull requests and issue reports
Detect clones (using iClones tool)
Detected clones (534,672 commits)
Build genealogies (using clone history)
Identifed genealogies (469,239 genealogies)
Idenitfy faults (SZZ) and collect features
1,817,482 data points for model training at a commit level
RQ1: LtR models to rank the fault-prone code clones
RQ2: Classification models to rank the fault-prone code clones
RQ3: Regression models to rank the number of fault changes in code clones
RQ4: Significant features to determine the rank of code clones

Fig. 2 Overview of our approach

2.6 Cross Validation

Cross-validation is a resampling technique used to evaluate machine learning models on limited data samples. Essentially, there are multiple folds (known as k-folds) where the k value is the number of folds. We choose the value of k as 10. The data is randomly divided into ten groups, and first nine folds are used for training and validation, and the last fold is used for testing. This process is repeated ten times using different folds each time for testing. In each fold for training, data is divided into a training set (t1) and validation set (v1). The model is trained on the data from the group's training set (t1) and then validated within the same group's validation set (v1). The performance results of each fold are aggregated.

3 Experiment Setup

This section describes the experiment setup used in this paper to examine the effectiveness of machine learning techniques at ranking code clones. Figure 2 presents the overview of our approach.

3.1 Project Selection

We use GHTorrent on the Google cloud[3] to extract all projects that have more than 1,000 commits, 1,000 issues, and 1,000 pull requests. We use such a high number of commits, pull requests, and issues to ensure that we have enough history of clone genealogies. We limit our study to Java and C projects. We focus on these two programming languages because code clones studies often use Java and C projects [3] [4]. Our selection criteria provide us with 66 Java and 29 C projects. Then, we discard the projects that are younger than five years (i.e., created after June 2015). A recent study on clone genealogies [3] suggests including projects with more than 100K source lines of code (SLOC). We remove the projects with less than 100K SLOC using the GitHub project SLOC calculator extension[4]. Furthermore, we remove the forked projects and the projects with less than 70% of code files (i.e., Java or C). The percentage

[3] `https://ghtorrent.org/gcloud.html`
[4] `https://github.com/artem-solovev/gloc`

of code files is calculated using the language information for each project in GitHub. After applying all the selection criteria, we obtain 34 Java projects and 18 C projects that are used in this study. The following steps are applied to both the Java and C projects. The details of the selected projects are provided in Table 6 available in appendix.

3.2 Building Clone Genealogies

The selected projects are all Git-based projects. Git provides multiple functions to extract the history of the projects. The history includes the renamed files, changed files, and changes made to each file using the `blame` function. We perform the following steps on each of the projects in our dataset. After downloading the repositories, we extract identifiers, committer emails, commit dates, and messages of each commit using the following command.

```
git log -- pretty=format:"%h,%ae, %ai, %s"
```

Detecting Code Clones. We use the latest version of the iCLONES clone detection tool[17] to identify the clones from the projects. We select iCLONES because it is recommended by Svajlenko *et al.* [43] who have evaluated the performance of 11 different clone detection tools. iCLONES uses a hybrid approach that includes the suffix tree and multiple revisions of the program to perform incremental clone detection. We use the settings recommended by Svajlenko *et al.* [43] since such settings are reported to achieve higher precision and recall values. We use the git checkout command to extract a snapshot of a project at a specific commit. We sort all commits chronologically and run the clone detection on each commit.

Extracting Clone Genealogies. Code clones may experience changes during the development and maintenance phases of the project. Such changes can be consistent or inconsistent based on a relative similarity score. If the copies of the clone are changed together, the clone pair is considered consistent. However, if only one copy is changed, the clone pair becomes inconsistent. An inconsistent clone pair can be later re-synchronized to make the clone pair consistent. A consistent clone pair can also diverge into an inconsistent state. The set of states (i.e., consistent and inconsistent) and the history of changes to any clone pair are known as clone pair genealogy. We identified the genealogies of all the clones in the studied projects as follows.

The iCLONES tool produces a list of clones that exist in a project at any specific commit. We link the clone pairs between each commit to create a set of genealogies. A change to a clone can affect its size, while a change to a file containing the clone can shift the position of the clone (i.e., changes its line numbers). To address this issue, we use the `git diff` command to detect all the changes to a specific file. We track the clone positional changes affected by the changes to the non-clone part of the file. We include only the changes to the clone contents rather than the clone line number since a shift in the line numbers does not change the state of the clone.

The clones are identified based on similarity. A pair of similar code snippets or a group of similar code snippets are reported by the clone detection tools. A number of recent studies [1][4][3][44] use clone pairs for performing different types of analysis on the code clones.

The prior studies show that clones need to maintain consistency in order to reduce the number of bugs. A clone pair is the fundamental unit for maintaining consistency. Moreover, the clone siblings in the clone group are not equally fault-prone [3], it is viable to perform fault proneness analysis on clone pairs, the most fundamental unit in clone analysis. It allows developers to pinpoint the problematic clone pairs in order to resolve the issues in the clone pairs instead of analyzing siblings in the whole clone group.

We build a clone genealogy for each clone pair detected by the iCLONES tool. We start by extracting the commit sequence of each project under study. We use the commit sequence to identify the modifications in the clone pairs of each commit. If a commit C2 changes a file that contains code in the clone pair, we use the `diff` command to compare the changes to a previous commit C1. If a clone snippet is changed in C2, we update the start and end line numbers of the clone from C2. To generate the mapping and check the modifications, we used a third-party python patching parser, called `whatthepatch` [10]. If the start or the end of the clone snippet is deleted, we move the clone line numbers accordingly to reflect the deleted lines.

A clone pair is said to be changed if a clone snippet is modified or deleted. When a clone pair is changed, we assess whether such a change is consistent or inconsistent. We verify this by searching the list of clone pairs generated by iCLONES. For each clone fragment having start and end line numbers, if another clone of this fragment is found in the tool results, a consistent state is assigned to the clone genealogy at that specific change point. Otherwise, an inconsistent state is assigned to the clone genealogy. It is important to note that a clone pair may undergo many changes where each change may have a different state. We repeat the process until one or both of the clones are deleted or until we reach the latest commit in the project.

The names of code files may also change during the development of the project. For each commit, we extract a file pair between the current and previous commits where an old file is deleted, and a new file is added. If the code similarity between the deleted file and the new file is more than 99%, we consider this as a renaming operation. A similar approach to identify renamed files is used by Barbour *et al.* [3]. We use the following command to extract the file pairs to detect renaming operations:

```
git diff [old-commit] [new-commit] --name-status -M
```

3.3 Detecting Faulty Clones

We use the SZZ algorithm [41] to identify the changes that introduced faults. First, we use the heuristic proposed by Fischer *et al.* [13], to identify fault fixing commits using a regular expression. The regular expression identifies the bug-ID in the commit messages. If a bug-ID appears in the commit message, we map the commit to the bug as a bug-fixing commit. Second, we mine the issue reports of each project from GitHub. For the issues that are closed, we identify if there are any pull requests associated with such issues. If there is a pull request associated with an issue, we identify all the commits included in the pull request and map the commits to the issue as a bug-fixing commit.

Once we have a list of all bug-fixing commits, we use the following command to identify all the modified files in each commit.

```
git log [commit-id] -n 1 --name-status
```

We consider only changes to code (i.e., Java and C) files in a commit. For all changes to a specific file of a bug-fixing commit, we use the following git blame command to identify all the commits when the same snippet is changed.

```
git blame -L startLineNumber,endLineNumber filePath
```

The *startLineNumber* and *endLineNumber* are the start and end of the code clone under consideration while *filePath* is the file name where code clone exists. We consider such commits as the "candidate faulty changes". We exclude the changes that are blank lines or comments. Finally, we filter the commits that are submitted before the creation date of the bug reports. We then check whether the commits identified as bug-inducing commits include clone pairs. If a clone snippet is included in the bug-inducing commits, we label the clone change as "buggy".

The buggy commit is a commit in which a fault introducing change is made. We only associate a commit to the genealogy if there is a change within the cloned lines of code. Once a buggy change is introduced via a buggy commit, a code clone becomes buggy at that specific change instance. If any future changes are made to fix the bug, the clone again becomes bug-free. To summarize, the clone genealogy can experience different state changes from buggy to bug-free and vice versa.

In our dataset, we treat each change in a code clone as a separate data point. For example, if there are 10 changes in a clone genealogy for any clone pair, there are 10 different data points for this instance. It essentially means that we are labelling the clone pair as faulty in any of the specific commit. Let us assume that a bug introducing change is in the third commit and bug fixing change happens in the fifth commit. The clone pair would be labelled as faulty in only the third and the fourth commit and would be fault-free in the fifth commit and on-wards unless a new bug happens in the genealogy.

During the lifetime of a clone genealogy, there can be buggy changes and subsequent bug-fixing changes later in the lifetime of the clone genealogy. In our dataset, we observe that more than 60% of the buggy clone changes are fixed at some points in the later stage of the project while the rest of the clone genealogies remain buggy. It is also interesting to note that while the clone genealogy is in a buggy state, more than 90% of the clone pairs are inconsistent. It means that both copies of the code clones are not changed together in such cases. To calculate the above-mentioned statistics, we identify the bugs in the code clones and match them with the state of the code clones (more details in the replication package). To identify the clone genealogies with the bug fixing changes, we measure if a code clone has a bug at any stage of the lifetime and later the bug is fixed.

3.4 Collecting Features

For each instance in a clone pair genealogy, we extract multiple features that may help rank fault-prone code clones. Some of the computed features were used in a prior study by Barbour *et al.* [3]. The features are divided into four categories; product, process, genealogy, and user metrics. Table 1 presents the description of our collected features.
Removing Correlated and Redundant Features. We remove the redundant features to avoid the possibility of having correlated features interfering in the interpretation of our models [40]. We use the Spearman rank correlation test with a cut-off value of 0.7 to identify the redundant features [20]. To construct the hierarchical overview of inter-feature relationships, we run the `varclus` function from the R package `Hmisc` [22]. The results indicate that: 1) clone age in days (*CAgeDays*) and clone age in commits (*CAgeCommits*) are correlated; and 2) submitted pull requests (*NFixPR*), accepted pull requests (*NAccPR*), and rejected pull requests (*NRejPR) are correlated.* We therefore retained *CAgeCommits* and *NFixPR*, and removed the rest of the redundant features from our model training step.

4 Results

In this section, we present the results of all of the research questions. We discuss motivation, approach and findings for each of the research question.

4.1 RQ1: Can we use learning-to-rank (LtR) algorithms to effectively rank fault-prone code clones?

Motivation. Developers in software projects aim to make the best use of their available time and resources. The maintenance of code clones can be a hectic task, especially in large software systems, as the number of clones can be immense. To identify the code clones at each commit level, developers can run a clone detection tool that provides developers with the list of clones at a specific commit. To identify clones that need to be maintained in priority, they often have to browse through a long list of clone candidates, which is time-consuming. If the risk of each code clone could be assessed accurately through an automatic process, we could save developers some precious time. Although previous studies [3] have examined the fault-proneness of different clone evolutionary patterns, to the best of our knowledge, they did not prioritize clones

Table 1 The calculated features for the clone pair genealogies

Metric	Description
	Product Metrics
CLOC	The number of cloned lines of code.
CPathDepth	The number of common folders within the project directory structure.
CCurSt	The current state of the clone pair (consistent or inconsistent).
FChgFreq	The average of the file changes experienced by clone pair.
CAgeCommits	The clone age in terms of the number of commits.
CAgeDays	The clone age in terms of the number days.
CSib	The number of siblings of the clone pair at any specific commit.
	Process Metrics
EFltDens	The number of fault fix modifications to the clone pair since it was created divided by the total number of commits that modified the clone pair.
TChurn	The sum of added and the changed lines of code in the history of a clone.
TPC	The total number of changes in the history of a clone.
NumOfBursts	The number of change bursts on a clone. A change burst is a sequence of consecutive changes with a maximum distance of one day between the changes.
SLBurst	The number of consecutive changes in the last change burst on a clone.
CFltRate	The number of faulty modifications to the clone pair divided by the total number of commits that modified the clone pair.
	Genealogy Metrics
EEvPattern	One of the *SYNC, DIV, INC, LP,* or *LPDIV* clone evolutionary patterns [3]
EConChg	The number of consistent changes experienced by the clone pair.
EIncChg	The number of inconsistent changes experienced by the clone pair.
EConStChg	The number of consistent change of state within the clone pair genealogy.
EIncStChg	The number of inconsistent change of state within the clone pair genealogy.
EFltConStChg	The number of re-synchronizing changes (i.e., *RESYNC*) that were a fault fix.
EFltIncSChg	The number of diverging changes (i.e., *DIV*) that were a fault fix.
EChgTimeInt	The time interval in days since the previous change to the clone pair.
UUsers	The number of unique committers in the clone genealogy.
	User Metrics
CommiterExp	The experience of committer (i.e., the number of previous commits submitted before a specific commit.)
NFixPR	The number of pull requests submitted by a specific committer.
NRejPR	The number of pull requests rejected for a specific committer.
NAccPR	The number of pull requests accepted for a specific committer.
Contributor	Core, if number of commits by a specific committer is higher than the average commits by contributors over the past months otherwise Casual.
CCurFile	The total number of changes to current file by a specific committer.
OChgRatio	The number of commits in the genealogy by a specific committer.

for maintenance. In this research question, we examine the possibility of leveraging LtR algorithms to provide developers with a ranked list of code clones based on the fault-proneness of the code clones. This ranked list can help developers make informed decisions and better utilize their time and resources by focusing on the most fault-prone clones. We select LtR algorithms, because they have been successfully used in previous studies to rank bug reports [57], web services [49], and pull requests [56].

Approach. We use three different types of datasets to train and test the LtR models: a) data from the whole lifetime of the projects (Model$_{all}$); b) data from the early phase of the projects (Model$_{early}$); and c) data from the mature phase of the projects (Model$_{mature}$). The definition of the Model$_{early}$

and Model$_{mature}$ are the same across all the research questions. Model$_{early}$ is used to model the early phase that is a project tenure during the start of the project while Model$_{mature}$ is used to model the mature phase that corresponds to the latest changes to a project.

To select the data for Model$_{early}$ and Model$_{mature}$, we proceed as follows. We subtract the latest commit date from the initial commit date of each project. The difference in the number of days is then converted to four quantiles. We use code clone data mentioned in Table 2 of the commits from the first (1^{st}) quantile of each project for Model$_{early}$ and code clone data mentioned in Table 2 of the commits from the fourth (4^{th}) quantile for Model$_{mature}$. A similar approach of time-based slicing of the data has been used in previous studies to slice Stack Overflow data [48], and Gitter Chatroom data [12].

To assess the effectiveness of LtR algorithms for clone ranking, we train the LtR models described in Section2.2 on our dataset of clones extracted from 52 projects and assess their effectiveness at ranking the clones. A LtR model is trained using a set of documents (d_1, d_2, ... ,d_n) for a set of queries (q_1,q_2,....,q_n). In our case, the documents are the code clone instances at any commit level, and the query is ranking all the code clones in the commit. A commit **C** can contain multiple code clones, including their history, and a single code clone can be a part of multiple commits as well but at a different time in their lifetime. The LtR algorithm computes the relevance between the documents (i.e., code clones) and the query using the features (defined in Section 3.4), and outputs a ranked list of code clones at each commit level. The most risky code clones in terms of fault-proneness appear at the top of this ranking. Developers can therefore prioritize them for maintenance. To assess the performance of the trained learning-to-rank models, we use binary relevance (MAP) and graded relevance (NDCG) evaluation metrics. We select these two metrics because we have binary labels in our models, i.e., 1 for the code clones that are fault-prone and 0 for the code clones that are not fault-prone. In addition to the aforementioned metrics, we also calculate the precision of the trained models through cross-validation.

The data related to the evolution of the software systems is time-sensitive [33], and this time sensitivity should be considered while evaluating the models trained on such data. Since code clones go through multiple changes during the evolution of the system, when building ranking models, we have to ensure that they are trained only on past code change data. To ensure this, we sorted our data based on the commit date, and for every project individually, we split the data into training and testing sets. It is important to note here that a code clone in the training set can have a commit date that is later than the commit dates of clones in the test set only if the clone is from a different project. In other words, we focus on the correct sorting of the clones within a project. The code clones that belong to the same project always have higher dates in the test set than in the training set. The time-sensitive nature of the data is carefully considered while conducting 10-fold cross-validation.

In this paper, we only identify clones within projects and perform the analysis as each project in our dataset has its own uniqueness. For example, they originate from different application domains, have their own development style, and follow various management policies [25].

Results.

LightGBM and XGBOOST perform better among the learning-to-rank techniques when ranking code clones based on their fault-proneness. Table 2 summarizes the results for the five algorithms and the two frameworks used to perform ranking. The models are evaluated using three LtR evaluation metrics (i.e., precision, MAP, and NDCG). LightGBM and XGBOOST outperform the pairwise and listwise algorithms and achieve a precision of 0.72 (Model$_{all}$). Overall, we can conclude that learning-to-rank algorithms do not perform well on clone ranking tasks, since developers would have to stiff through a long list of false positives if they were to adopt any of these models. We attribute this weak performance to the nature of the objects that are being ranked. In fact, multiple code clones at the commit level can have the same risk of fault, while LtR models are known to perform well when documents have clear distinctive ranks[56]. In our models, the dependant variable is a Boolean variable indicating whether or not a clone pair experienced a fault-inducing change. Our query is to rank all the pairs in a commit. For example, if there are three different clone pair changes ($c1, c2, c3$) in a specific commit T, there can be three possibilities.

– None of the changes is fault-prone, (0,0,0);
– all the changes are fault-prone, (1,1,1);
– there are faulty as well as non-faulty code clones, e.g., (0,1,0).

LtR performs well when the documents in the training data has a clear distinctive rank, e.g., (0,1,2). Moreover, the performance of the model increases with the availability of historical data. In the early phase of the projects (i.e., Model$_{early}$), the performance of the model is lower. When more clone history data is available (i.e., **Model**$_{mature}$), the models perform slightly better.

Summary of RQ1

Among the studied learning-to-rank (LtR) algorithms/frameworks LightGBM and XGBoost achieve better results. Overall, learning-to-rank (LtR) algorithms/frameworks perform moderately due to the nature of the code clone representation at a commit level. The performance of the models increases with the level of maturity of the projects, possibly due to the availability of more information about the history of the code clones.

Table 2 Results of evaluation metrics for LtR algorithms for Java and C projects

Type	**Algorithms**	**Java Projects**			**C Projects**		
		Precision	**MAP**	**NDCG**	**Precision**	**MAP**	**NDCG**
		Model$_{all}$					
Pairwise	RankBoost	0.56	0.05	0.17	0.55	0.06	0.15
	RankNet	0.58	0.07	0.20	0.60	0.05	0.17
Listwise	LambdaRank	0.61	0.09	0.22	0.63	0.09	0.20
	LambdaMart	0.60	0.11	0.23	0.61	0.12	0.21
	Random Forest	0.67	0.13	0.26	0.66	0.15	0.24
Framework	XGBOOST	0.71	0.13	0.27	0.67	0.17	0.26
	LightGBM	0.72	0.14	0.27	0.69	0.14	0.26
		Model$_{early}$					
Pairwise	RankBoost	0.52	0.03	0.15	0.52	0.03	0.12
	RankNet	0.54	0.04	0.17	0.57	0.04	0.15
Listwise	LambdaRank	0.59	0.08	0.21	0.61	0.08	0.18
	LambdaMart	0.57	0.09	0.20	0.57	0.10	0.19
	Random Forest	0.64	0.11	0.22	0.63	0.14	0.21
Framework	XGBOOST	0.68	0.11	0.25	0.66	0.16	0.24
	LightGBM	0.69	0.13	0.26	0.67	0.11	0.24
		Model$_{mature}$					
Pairwise	RankBoost	0.57	0.05	0.18	0.56	0.07	0.15
	RankNet	0.57	0.08	0.20	0.61	0.04	0.19
Listwise	LambdaRank	0.62	0.08	0.21	0.64	0.07	0.19
	LambdaMart	0.62	0.11	0.24	0.63	0.14	0.22
	Random Forest	0.68	0.14	0.24	0.67	0.13	0.23
Framework	XGBOOST	0.72	0.14	0.28	0.69	0.18	0.27
	LightGBM	0.73	0.19	0.29	0.71	0.17	0.28

4.2 RQ2: How well can classification algorithms rank fault-prone clones?

Motivation. In RQ1, we used the learning-to-rank (LtR) machine learning algorithms to rank the clone pairs based on their fault-proneness. However, these algorithms only achieved moderate performance (i.e., the precision of 0.72 at best). Hence, in this research question, we examine the possibility of using classification algorithms to achieve better results. Using historical information about the clone pairs, we use classification algorithms to classify them as buggy or not buggy. The probability that a clone pair belongs to the faulty class is used for ranking. These ranks can be used by developers to prioritize clones for maintenance.

Approach. Similar to **RQ1**, we use three different types of the datasets to train and test the classification models: a) data from the whole lifetime of the projects (Model$_{all}$); b) data from the early phase of the projects (Model$_{early}$); and c) data from the mature phase of the projects (Model$_{mature}$). We use the 28 features (related to code clones) described in Section 3.4 to train the models. We train and evaluate the models as follows.

Step 1: Training Phase

a) Data for Training and Testing. The data for the three different models are selected in different ways. Data for Model$_{all}$ contains the data from the whole lifetime of the projects. There are 1,789,457 data points from the Java projects and 1,154,673 data points from the C projects. To select the data for Model$_{early}$ and Model$_{mature}$, we proceed as follows. We subtract the latest commit date from the initial commit date of each project. The difference in the number of days is then converted to four quantiles. We used code clone data of the commits from the first (1^{st}) quantile of each project for Model$_{early}$ and code clone data of the commits from the fourth (4^{th}) quantile for Model$_{mature}$. A similar approach of time-based slicing of the data has been used in previous studies to slice Stack Overflow data [48], and Gitter Chatroom data [12]. For Model$_{early}$, we obtained 395,256 rows of data for the Java projects and 156,781 rows of data for the C projects. For Model$_{mature}$, we obtained 725,158 rows of data for the Java projects and 475,317 rows of data for the C projects.

b) Problem Formulation. The dependent variable of our model takes the value 0 when a clone pair is not buggy and 1 when the clone pair is buggy. We use Naive Bayes, Logistic Regression, Random Forest algorithms to classify the clone pairs. We select these algorithms because they are commonly used in classification problems. In addition, we also use the LightGBM and XGBOOST's classifier functions to classify the code clones pairs.

Step 2: Evaluation Phase

We used 10-fold cross-validation to evaluate the performance of the models. We created time-consistent folds as described in Section 2.6. We sorted commit data using commit dates and ensured that folds on which models are tested always contain data that is posterior to the data on which the models were trained. Within each fold, we ensure that validation sets always contain data that is posterior to the data on which the models were trained. This time consistency ensures that we are not predicting past data using future data. We assessed the performance of the models using the metrics described in Section 2.

Results

Random Forest achieves the highest AUC Score when classifying clone pairs based on their fault-proneness. Table 3 presents the results for three classification algorithms and two machine learning frameworks. For all the evaluation metrics, Random Forest achieves the highest scores for both Java and C projects. The reported values are the mean values obtained over the 10-fold cross-validation. The performance of Logistic Regression, LightGBM, XGBOOST, and Random Forest are superior to the maximum performance values obtained in **RQ1** when using LtR algorithms. In terms of the execution time, LightGBM is 5x faster than the Random Forest.

Random Forest also achieves the best performance when models are trained using only data from the early phase, or only data from the mature phase. The results of Model$_{early}$ and Model$_{mature}$ presented in Table 3 show that models trained exclusively with data collected during the early phase of the projects (i.e., Model$_{early}$) achieve lower performance

Table 3 10-fold cross-validation results of three classification algorithms and two frameworks, evaluation metrics precision, recall, accuracy, AUC, and F1-score. Java and C projects results presented separately.

	Java projects					**C projects**				
Evaluation Metrics	**Precision**	**Recall**	**Accuracy**	**AUC**	**F1-Score**	**Precision**	**Recall**	**Accuracy**	**AUC**	**F1-Score**
Model$_{all}$										
Logistic Regression	0.80	0.58	0.80	0.75	0.67	0.78	0.62	0.79	0.74	0.66
Naive Bayes	0.49	0.79	0.64	0.67	0.60	0.52	0.78	0.64	0.66	0.63
LightGBM	0.91	0.78	0.90	0.87	0.84	0.90	0.75	0.89	0.87	0.85
XGBOOST	0.91	0.78	0.89	0.87	0.84	0.89	0.77	0.90	0.88	0.86
Random Forest	0.98	0.96	0.98	0.97	0.97	0.97	0.96	0.94	0.96	0.96
Model$_{early}$										
Logistic Regression	0.70	0.37	0.76	0.65	0.49	0.71	0.37	0.77	0.64	0.52
Naive Bayes	0.56	0.33	0.71	0.61	0.41	0.55	0.33	0.75	0.60	0.43
LightGBM	0.92	0.76	0.91	0.86	0.83	0.90	0.77	0.90	0.85	0.84
XGBOOST	0.91	0.78	0.91	0.87	0.84	0.88	0.77	0.88	0.86	0.82
Random Forest	0.96	0.90	0.96	0.94	0.93	0.95	0.89	0.94	0.94	0.92
Model$_{mature}$										
Logistic Regression	0.64	0.16	0.89	0.57	0.26	0.63	0.18	0.90	0.58	0.29
Naive Bayes	0.20	0.84	0.58	0.69	0.32	0.25	0.85	0.60	0.70	0.35
LightGBM	0.98	0.79	0.97	0.90	0.88	0.98	0.77	0.96	0.89	0.89
XGBOOST	0.99	0.83	0.98	0.92	0.90	0.97	0.82	0.96	0.90	0.91
Random Forest	0.99	0.95	0.99	0.97	0.97	0.98	0.94	0.97	0.96	0.97

than the models trained using data from all the evolutionary history of the projects (i.e., Model$_{all}$). However, models trained using only data from the mature phase of the project (i.e., Model$_{mature}$) achieve higher performance than Model$_{all}$. This result suggests that models are able to more precisely capture the risk of faults of code clones when more information about the evolutionary history the project is available. However, the recency of the information is important (as shown by the high performance of Model$_{mature}$). During maintenance activities, development teams looking to prioritize clones should pay attention to give the models as much recent information about the clones as possible.

Summary of RQ2

Random Forest achieves the best AUC when classifying the clone pairs based on the fault-proneness of the code clones. Random Forest also achieves high scores of evaluation metrics in early and mature phase data, while the mature phase model performs better than the early and all phase models.

4.3 RQ3: Can we use regression algorithms to predict the proportion of faulty changes in code clones and effectively rank fault-prone clones?

Motivation. In this research question, we explore the possibility of using regression algorithms to rank code clones. Specifically, we use regression models to predict the ratio of future faulty changes for each clone pairs and use this information to rank the clone pairs. Clone pairs that are predicted to experience the highest ratio of faulty changes are ranked at the top, i.e., they are considered to be more risky than clone pairs predicted to experience a lower ratio of faulty changes. Similarly to **RQ2**, we predict the ratio of faulty changes at different stages of the development process (i.e., early and mature).

Approach. Similar to **RQ2**, we use three different types of the dataset to train the regression models: a) data from the whole lifetime of the projects (Model$_{all}$); b) data from the early phase of the projects (Model$_{early}$);, and c) data from the mature phase of the projects (Model$_{mature}$).

To train the models, we computed the values of 28 features that belong to four categories of clone-related metrics: process, product, genealogy, and user. Table 1 provides a description of the features. One common problem in regression analysis is multicollinearity, which occurs when two or more independent variables are highly correlated. Although multicollinearity may not affect the accuracy of the model much, it causes imprecise estimates of the coefficient values of the model, which prevents distinguishing precisely between the individual effects of the different independent variables on the dependent variable. Variance Inflation Factor (VIF) is used to determine the level of multicollinearity for the regression problems [8][50]. As we have used the regression analysis in RQ3 and RQ4, the VIF approach is used to identify the level of multicollinearity among the features from the dataset. The varclus is used to extract the correlated features from the dataset for RQ1 and RQ2. Following standard guidelines [8] [50], we retained in our models only features for which the VIF is under 5. The following paragraphs provide more information about the training and evaluation phase of our models.

Step 1: Training Phase

a) Data for Each Model. As explained in the previous research question, there are 395,256 data points for the Java projects and 156,781 data points for the C projects in Model$_{early}$. For Model$_{mature}$, there are 725,158 data points for the Java projects and 475,317 data points for the C projects. Model$_{all}$ includes all the data points collected by using our data selection criteria described in Section 3.

b) Problem Formulation. The independent variables (X) of our models are the features described in Table 1. The dependent variable (Y) is the ratio of faulty changes for a clone pair. In our case, the ratio of faulty changes is expressed as the ratio of faulty commits in which a code clone experienced a change. We calculate the ratio of fault-prone changes using the SZZ approach as explained in Section 3. The ratio of faulty changes is calculated using the equation 9.

$$RatioofFaultyChanges = \frac{\text{Number of fault-prone changes}}{\text{Total number of changes}} \tag{9}$$

It is important to note here that we omit the *CFltRate*(fault-prone modifications) feature in this research question because it is related to our dependant variable. Once we predict the ratio of faulty changes for each clone pair at the commit level, we rank the code clones from high to low based on their estimated ratio of future faulty changes. A ratio of faulty changes of 0 means that the clone pair is not expected to experience a fault-fixing change in the future, while a ratio of 1 means that all future changes on the clone pair are predicted to be faulty.

We use a mixed-effect model [42], to take into account the context of the project in our analysis of fault-prone code clones. A mixed-effect model presents the significant features while keeping in view the context during the model training. The mixed-effect model consists of two types of features, explanatory features, and context features. The explanatory features (i.e., process, product, genealogy, and user) are used to explain the data, while context features (i.e., project) are used to determine the effect of explanatory features. The mixed-effect model is able to show the relationship between the outcome (i.e., the ratio of faulty changes) and the explanatory features while taking into consideration the context features (i.e., project).

We use a linear regression (mixed-effect) as our baseline model (as used in prior studies [12] [20] for software engineering problems) to determine the ratio of faulty changes for a clone pair. We also use ridge regression and lasso regression algorithms; which are commonly used to train regression models. Finally, we use Random Forest, regressor function of XGBOOST, and regressor function of LightGBM. Overall, we experimented with four regression algorithms and two frameworks. We use the implementation of `scikit-learn` [35] for the four algorithms and for two frameworks (i.e., XGBOOST[5] and LightGBM[6]), we use their open source implementation available on GitHub.

Step 2: Evaluation Phase

Similarly to **RQ2**, we used a 10-fold cross-validation approach to evaluate the models, constructing time-consistent folds as described in Section 2.6. We sorted commit data using commit dates and ensured that folds on which models are tested always contained data that are posterior to the data on which the models were trained. We also ensured that validation sets always contained data that are posterior to the data on which the models were trained. These time-consistent folds help ensure that we are not predicting past data using future data.

[5] https://github.com/dmlc/xgboost
[6] https://github.com/microsoft/LightGBM

From the results of the cross-validation approach, we can interpret whether the model is over-fitting. If the training RMSE/MAE and test RMSE/MAE has significant differences, it indicates that the model is over-fitting.

Results

LightGBM outperforms the other algorithms/frameworks in ranking code clones based on their estimated ratio of future faulty changes. Table 4 presents the results of the cross-validation for the four algorithms and two frameworks. The models are trained separately for Java and C software projects. The LightGBM algorithm is able to achieve an R-Squared score ranging from 0.87 (for C projects) to 0.89 (for Java projects) for Model$_{all}$, which is much better than the baseline Logistic Regression (0.70 for C and 0.71 for Java projects). The test and train RMSE and MAE are also better for LightGBM than the other studied algorithms. The performance metrics of XGBOOST and Random Forest show that they outperform the baseline and achieve a performance close to that of the top performer LightGBM. It is important to note here that the train and test RMSE and MAE scores are uniform, which is one of the indicators that the model is not overfitting. The regression results are significantly better than the learning-to-rank(LtR) results while classification models seem to be more effective. However, the classification approach is a more simplistic approach which only provides developer with the information on whether a clone pair would have a fault. The regression approach provides the ratio of faulty changes in a clone pair and achieve good results. Developers can use the approach based on their specific need and improve the maintenance of the code clones.

XGBOOST and LightGBM can be used in the early and mature phases of software projects to determine the rank of the code clones. Table 4 shows the evaluation metrics results for Model$_{early}$ and Model$_{mature}$. XGBOOST achieved an R-squared value ranging from 0.83 (for C projects) to 0.84 (for java projects) for the Model$_{early}$. The evaluation metrics results for the early phase is lower than that of the Model$_{all}$ which is understandable because the history of the code clone is not developed in the early phase. However, it is encouraging to see that Model$_{mature}$ is able to outperform Model$_{all}$. This is similar to the results obtained with classification algorithms. A possible explanation for this phenomenon is the recency of information used in Model$_{mature}$. The occurrence of faults in matured clone pairs is likely to be more predictable than the occurrence of faults in newly created clone pairs, and recent information about the clone pairs are better predictors of fault occurrences than older information. During maintenance activities, developers can select the appropriate model (by taking into account the level of maturity of the project) to predict the risk of faults in code clones contained in their

Table 4 10-fold cross-validation results of four regression algorithms and two frameworks, evaluation metrics R-squared, RMSE, MAE. Java and C projects results presented separately.

	Java projects					**C projects**				
Evaluation Metrics	**R-squared**	**Train RMSE**	**Test RMSE**	**Train MAE**	**Test MAE**	**R-squared**	**Train RMSE**	**Test RMSE**	**Train MAE**	**Test MAE**
Model$_{all}$										
Linear regression (mixed-effect)	0.78	23.4	24.5	42	44.2	0.68	23.5	25.8	43.8	45.7
Ridge regression	0.79	21.2	22.6	37.2	38.9	0.77	22.5	23.7	38.6	39.9
Lasso Regression	0.78	18.6	19.5	35.6	36.8	0.77	19.2	21.5	37.2	39.2
Random Forest	0.86	9.5	10.3	18.6	20.3	0.85	10.6	12.2	18.3	19.7
XGBOOST	0.86	5.2	5.8	13.8	15.1	0.86	5.4	6.3	14.2	15.7
LightGBM	0.87	4.7	4.95	11.1	12.3	0.86	4.9	5.6	11.4	12.8
Model$_{early}$										
Linear regression (mixed-effect)	0.77	24	25.7	44	46.7	0.65	24.3	26.5	44.5	47.2
Ridge regression	0.76	22.4	23.9	38.6	39.5	0.75	24.8	26.7	39.5	42.1
Lasso Regression	0.76	19.5	21.3	36.1	38.6	0.74	23.5	24.9	38.4	40
Random Forest	0.82	10.5	11.7	19.3	21.8	0.82	11.6	13.1	19.5	21.6
XGBOOST	0.84	5.9	7.1	14.8	16.4	0.83	6.2	7.6	15.3	16.9
LightGBM	0.84	5.8	7.3	14.5	16.5	0.82	6.1	7.9	16.5	17.7
Model$_{mature}$										
Linear regression (mixed-effect)	0.80	22.7	23.9	41.6	44	0.0.71	22.7	25.3	43	45.2
Ridge regression	0.79	20.9	21.7	37.1	39.2	0.78	21.9	23.4	38.1	39.5
Lasso Regression	0.80	19.3	21	36.3	37.5	0.79	20.2	21.8	37.5	39.8
Random Forest	0.87	8.8	10.1	17.5	19.3	0.86	9.7	11.3	17.9	19.2
XGBOOST	0.89	4.9	5.4	12.3	13.4	0.88	5.1	5.8	13.5	14.6
LightGBM	0.90	4.1	4.6	9.7	10.9	0.89	4.3	5	10.3	11.8

system. The top ranked code clones can be fixed first, to prevent future clone related faults.

Summary of RQ3

LightGBM outperforms the studied algorithms/frameworks when trained for ranking code clones based on the ratio of faulty changes. However, LightGBM ad XGBOOST both can be used in the early and mature phases of the project as they perform well. The mature phase performance from the previous two research questions also holds in this context.

4.4 RQ4: Which metrics are significantly correlated to the risk of faults in code clones?

Motivation. In RQ2 and RQ3, we trained machine learning models to rank code clones at the commit level, using 28 features from four different categories. The detection of code clones, the identification of clone genealogies, and the computation of the features of the models can be costly. In this research question, we examine the importance of each feature used in our models to identify the subset of features that are the most important for predicting fault occurrence in clone pairs. It is important to note here that we only identify significant features for our regression approach (RQ3),since it provides the ratio of faulty changes and achieves good performance. Developers can use similar steps to identify significant features in the learning-to-rank (LtR) and classification approaches. The identified features can provide quick guidance (before using machine learning models trained in previous research questions) to developers, helping them to prevent fault-occurrence in code clones.

Approach. In this section, we describe the approach followed to identify the significant features in predicting the rank of the code clones, based on their fault-proneness. We computed features following the steps described in Section 3.4. The data for all the three models (i.e., Model$_{all}$, Model$_{early}$, and Model$_{mature}$) are collected using the approach described in RQ2. We also identify the level of multicollinearity using VIF as mentioned in RQ3.

Building and Analyzing the Mixed-Effect Model. Because the studied software projects belong to different domains and different programming languages, the behavior of code clones in these projects is likely to be different. To take into account the context of the project in our analysis of fault-prone code clones, we use a mixed-effect model [42] as already explained in RQ3.

It is important to note that we build three different mixed-effect models (i.e., Model$_{all}$, Model$_{early}$, and Model$_{mature}$) for each of our three datasets. We use the `glmer` function of the R package `lmer`[5] to construct mixed-effect models. The discriminative power of the model measures the ability of the model to distinguish values 0 and 1 of the dependant variable (i.e., predicting the number of faulty changes of the code clones). We use the Area Under

Curve (AUC) [19] to determine the discriminative power of the model. We use Receiver Operator Curve (ROC) to plot the true positives against the false positive for different thresholds. The value of AUC ranges from 0 to 1, 0 being the worst performance, 0.5 being the random guessing performance, and 1 being the best performance. [19]

To understand the impact of explanatory features, we use Wald statistic [28] to estimate the relative contribution (X2). A higher value of (X2) shows the high impact of the feature on the performance of the model [20]. We use the R package Car [15] which provides the implementation of `anova` to calcualte wald X2.

Results

The increased involvement of developers in the clone genealogy can have an effect on the fault proneness of the clone pair. Table 5 presents the significance between the clone-related features and the number of faulty changes of the clone pairs. Due to the space constraint, Table 5 shows only the top five significant features for all three studied models. The AUC is measured to be 0.79 for the Model$_{all}$. However, detailed results of Model$_{all}$ (Table 7) , Model$_{mature}$(Table 8), and Model$_{early}$(Table 9) are available in the Appendix of this paper. The feature *unique users* accounts for the highest χ^2 in the mixed-effect model of Model$_{all}$, suggesting that the fewer is the number of developers changing a clone pair, the lower is the risk of faults. This result is understandable because if more developers make changes to a clone pair, they might not know about the other copies of the clone. Therefore, the clone copies can become inconsistent, and the probability of introducing a fault becomes higher than that of preserving the consistency of clones. In addition, *age_commits* is the second most significant feature. This essentially means that as more changes are made on a clone genealogy, the risk for fault increases.

In mature projects, the age of code clones (in terms of the number of commits) is correlated with the risk of faults. Table 5 shows that *age_commits* is the most significant feature for Model$_{mature}$. The AUC is measured to be 0.85 for the Model$_{mature}$, suggesting that, similarly to RQ2 and RQ3, the models achieve better performance when more historical information about the clones is available. Some code clones are changed a lot, while other experience significantly fewer changes. If a software project has an extensive code clone history, it is better to use Model$_{mature}$ to obtain better results. The *UnqUsers* (unique committers to a clone pair) is identified as the second most important feature in Model$_{mature}$. Hence, similarly to Model$_{all}$, the risk of fault introduction increases when multiple developers change a clone pair. Developers can focus on these two features (i.e., *age_commits* and *UnqUsers*) to reduce the risk of fault introduction in clone pairs.

The changes to code clones in the early phase of the software project should be performed by a fewer number of developers. As shown in Table 5 for Model$_{early}$, *UnqUsers* is the most significant feature in determining the rank of clone pairs based on fault-proneness. The AUC is

Table 5 Results of the mixed-effect model for Model$_{all}$, Model$_{mature}$, and Model$_{early}$. Sorted by χ^2 in decreasing order

. Top 5 features only

Factor	Coef.	χ^2	Percentage	$Pr(<\chi^2)$	Sign.$^+$	Relationship
			Model$_{all}$			
(Intercept)	$-4.653e^{+00}$	1238		$<2.2e^{-16}$	***	↘
UnqUsers	$1.617e^{-01}$	300315	64.53	$<2.2e^{-16}$	***	↗
age_commits	$4.907e^{-01}$	117350	25.22	$<2.2e^{-16}$	***	↗
FCngFreq	$-2.470e^{-03}$	20924	4.50	$<2.2e^{-16}$	***	↘
UChgRto	$2.668e^{-03}$	5917	1.27	$<2.2e^{-16}$	***	↗
age_days	$2.823e^{-04}$	4352	0.94	$<2.2e^{-16}$	***	↗
			Model$_{mature}$			
(Intercept)	$-6.772e^{+00}$	324		$<2.2e^{-16}$	***	↘
age_commits	$5.230e^{-01}$	11639	52.24	$<2.2e^{-16}$	***	↗
UnqUsers	$1.228e^{-01}$	2152	9.66	$<2.2e^{-16}$	***	↗
UFileChg	$-1.668e^{-03}$	1318	5.92	$<2.2e^{-16}$	***	↘
FCngFreq	$-1.185e^{-03}$	1248	5.60	$<2.2e^{-16}$	***	↘
age_days	$5.290e^{-04}$	1119	5.02	$<2.2e^{-16}$	***	↗
			Model$_{early}$			
(Intercept)	$-4.400e^{+00}$	103		$<2.2e^{-16}$	***	↘
UnqUsers	$1.638e^{-01}$	40938	52.42	$<2.2e^{-16}$	***	↗
age_commits	$4.420e^{-01}$	26567	34.02	$<2.2e^{-16}$	***	↗
CPathDepth	$1.082e^{-01}$	3762	4.82	$<2.2e^{-16}$	***	↗
cloc	$5.170e^{-03}$	3736	4.78	$<2.2e^{-16}$	***	↗
SLBurst	$-1.163e^{-01}$	430	0.55	$<2.2e^{-16}$	***	↘

$^+$Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

measured to be 0.77 for Model$_{early}$. If a clone pair is changed by multiple developers in the early phase of a project, the risk of fault introduction increases. To prevent the introduction of faults, developers can strive to keep the number of developers involved in changing cloned code to a minimum level. We suggest that any code clone changed by more than two developers should be carefully monitored. As observed in Model$_{early}$ results, *age_commits* is also the second most important predictor of future faults in clone pairs during the early phase of a project. A clone pair experiencing a higher number of changes at the start of the project should be monitored carefully. The feature *age_commits* has a higher importance in Model$_{early}$ in comparison to Model$_{all}$; which emphasizes that a higher number of changes to a clone pair at the start of the project is not recommended.

Summary of RQ4

When there are more developers changing a clone pair, the likelihood of the clone pair having fault in future increases significantly. As a project becomes mature, the age of code clones also show an association with the introduction of faults in the clone pair. At the start of the project developers should make sure that code clones are changed mostly by the authors of the clone pairs.

5 Implications

In this section, we discuss the implications of our results for the developers of open-source software projects. First, we will discuss the results of the machine learning approaches used to rank the code clones based on fault proneness. Then, we will specify specific guidelines for the stakeholders. The learning-to-rank(LtR) algorithms (RQ1) show moderate results due to the unavailability of the labeled ranked dataset. However, the classification algorithm (Random Forest) achieves better performance among the studied algorithms (RQ2). The developers can find this approach very effective in cases where they try to identify whether a clone would be fault-prone. In addition, if developers want to achieve a more accurate ranking by identifying the code clones that can have more fault-prone changes, they can use our regression approach (RQ3) that achieves higher evaluation scores using LightGBM. Based on the results from our research questions, we suggest the following guidelines to developers.

Developer can use the learning-to-rank (LtR) approach to rank code clones at a commit level. Our results suggest that LtR algorithms can be useful but have a lower accuracy than classification and regression algorithms. Developers who can label code clones with different ranks based on the associated faults can obtain better results. The developers can use different rankings based on their project needs. For example, developers can give a higher rank to functional bugs, a middle rank to non-functional bugs, and a lowest rank to non-faulty clones. It can be useful to rank the code clones on a different scale rather than just 0 and 1. A recent study [56] on pull requests uses a distinctive rank (0,1,2) on pull requests based on their priority and shows promising results. Developers can extract clone genealogies and calculate the features provided in Section 3.4 to train the LtR model using XGBOOST and LightGBM.

Developers can use classification models to rank code clones. Classification algorithms, Random Forest in particular achieves high efficiency in determining the probability of a code clone having a bug. The rank of the code clones using these probability values can help identify the most risky code clones at a commit level.

Developers can predict the number of faulty changes of the code clones using regression techniques. We use multiple regression techniques to estimate the risk of future faults in clone pairs. Our results show that LightGBM and XGBOOST achieve the best results. We ranked the code clones based on the number of faulty changes of the code clones, which developers can use to focus on top risky clones instead of going through all the code clones. Developers can choose the number of clones to refactor based on their resources and eventually, it would prevent future faults.

We provided specialized models for the early and mature lifetime of open-source software projects. In order to assist the new and old software projects efficiently, we trained two different models by explicitly choosing the data from these two stages of the software projects from our dataset. The specialized models can help developers in making an informed decision related to maintaining code clones at different stages of the development process.

We have identified significant features for developers to look out for when maintaining code clones. If the developers want to get a quick knowledge about the specific features related to code clones that they can focus on (if they are short on time). Developers must be cautious of too many changes by their peers to a particular code clone as it affects the number of faulty changes of the code clones. Moreover, it is essential to keep an eye on the code clones that experience many changes as they tend to be faulty in the future. We suggest that any code clone that is above the average age of code clones in the project should be monitored carefully. We also identify significant features (e.g., code clones in terms of the number of commit and unique committers in code clone history) for the new and old projects as well, that can aid the developers for their specific software projects.

6 Related Work

In this section, we summarize the existing literature related to the code clones. Specifically, we review research work related to code clone detection, fault-proneness of code clones, and analysis of the clone genealogies.

6.1 Code Clone Detection

Prior studies propose state-of-the-art clone detection tools that are able to identify code clones from large codebases. Göde *et al.* [17] propose an incremental clone detection tool (iClones) which detects clones based on the analysis of the previous revisions. iClones creates a mapping between the multiple revisions of code clones. iClones effectively provides the add/delete information of the code clones. Incremental detection of clones is helpful in evolutionary clone analysis and is fast as compared to other similar clone detection tools. Kamiya *et al.* [24] propose transformation rules and a token-based comparison clone detection technique that is optimized to achieve better performance. The technique converts the source code into tokens and, based on the pre-defined rules, performs the comparison. The clone detection technique is evaluated on four large-scale projects, and the results show that the tool is able to identify clones. Cordy *et al.* [9] propose a clone detection technique that uses a language-sensitive parsing and language-independent similarity analysis to identify near-miss clones. The approach is available as a simple command-line tool, and code directories are used as input while the output is available in a HTML and XML format. The approach is able to achieve high precision and high recall and is scalable to extensive systems.

Schwarz *et al.* [39] propose a set of lightweight techniques that can detect the code clone from the large codebase as well as across projects. The approach used a bad hashing concept to index the code clones. Squeaksource, a dataset of multiple projects with the project history, is used to evaluate the proposed lightweight techniques. The analysis shows that 22% of all type-3 clones could be missed if the analysis is performed on only the latest versions. Saha *et al.* [37] focus on the extraction and classification of near-miss code clone genealogies. Clone genealogy extraction starts by accepting multiple versions of a program, maps clone classes between the consecutive versions, and extracts how the cloned fragments are changed throughout the period. Finally, the approach identifies change patterns (i.e., added, deleted, modified) using a three-pass algorithm. Yang *et al.*[54] use machine learning models to classify the true clones based on the code clones detected by the clone detection tools. A web-based tool (FICA) is provided as a proof of concept to identify true clones. A user study is performed on interviewing 32 participants to evaluate the usefulness of the tool. The authors report 70% accuracy for the classification tool. Roy *et al.* [36] focus on identifying near-miss clones (i.e., where small to large changes have been made to the copied fragments of code). A hybrid approach is presented to detect clones, followed by the metamodel of the clone types. An empirical study of cloning in more than 20 open-source systems is performed for current clone detection techniques and tools.

The summary of the clone detection tool shows that iClones perform better in identifying code clones when there are multiple revisions (i.e., commits) of the projects. Therefore, we use iClones to identify the clones from our studied projects.

6.2 Fault-Proneness of Code Clones

Inconsistent changes of code clones can lead to the introduction of bugs in the project. It is vital to effectively make changes to the copies of clones to avoid the risk of bugs. The following studies focus on the impact of bug propagation and the fault-proneness of different types of clones.

Mondal *et al.* [31] present an empirical study to understand the intensity of bug-propagation through code cloning. The empirical analysis shows that up to 33% of the code clones can be related to bug prediction, provided that code clones experience bug fix changes. Near-miss clones (i.e., Type 2 and Type 3 clones) have a higher chance of being involved in the bug-propagation than the identical clones (i.e., Type 1). Xie *et al.* [52] present a study of fault-proneness of Type-3 code clones in the evolving software. The study analyzes three long-lived software systems APACHE-ANT, ARGOUML, and JBOSS, written in JAVA. The clone genealogies are build using the NICAD clone detection tool to examine two evolutionary processes: 1) how clone types are mutated in a system; and 2) how code clones are migrated in a project. Li *et al.*[30] propose a keyword based approach to identify the buggy code clones. The bug reports are evaluated and are linked with the code clones using the keyword approach to identify the buggy code clones. The bugs related to code clones comprise of 4% of overall bugs in the studied systems. The approach is available as a tool that can be used to identify buggy code clones. Saha *et al.* [38] perform an exploratory study about the evolution of Type-1, Type-2, and Type-3 clones in six open source projects written in two different languages. Results show that a considerable number of type-1 and type-2 clones change to type-3 clones during evolution. Although the life span of type-3 clones is similar to type-1 and type-2 clones, it is essential to manage the type-3 clones to limit their negative impact. Barbour *et al.* [4] examine the characteristics of late propagation of code clones in two software systems. Late propagation introduces the concept of making changes to one clone, and the other is changed afterward in the next revisions; this evolutionary pattern can introduce faults. The study defined eight different types of late propagation and compared them with other forms of clone evolution.

In general, the studies related to the fault-proneness of code clones includes late propagation, code clones types analysis related to bug prediction, and bug prediction for code clones provided that code clones experience faulty changes. To the best of our knowledge, the fault-proneness of the code clones has not been used to rank the code clones in order to improve the maintenance of such code clones. We use fault-proneness information to rank code clones at the commit level, so that fault-prone code clones can be fixed first.

6.3 Analysis of Clone Genealogies

The clone genealogies need to be built first using the history of code clones before the analysis of code clones. Barbour *et al.* [3] investigate six different evolutionary patterns using the clone genealogies extracted from four open-source Java systems. The analysis uses the clone genealogy information to identify the fault-prone nature of clone pairs based on the evolutionary patterns of the clone pairs. The results show that including the clone genealogy information can increase the identification power of fault prediction models. Zhang *et al.* [55] use clone genealogy information to predict the consistency-maintenance of code clones. The study identifies the code clones with consistency issues earlier in the lifetime in order to improve the maintenance of such code clones. The approach can predict the consistency-maintenance of the code clones. Thongtanunam *et al.* [46] investigate the life of code clones using the clone genealogy information from the code clones. The approach used multiple features to train a Random Forest classifier to determine whether a code clone would be short-lived. Garg *et al.* [16] propose a clone ranking approach for better clone management. The paper prioritizes the results of clone detection tools by finding out the maintenance overhead using the size of code clone, MCC complexity and frequency of clone results. Different weights are provided to the three selected features. The CCFinder tool [24] is used to detect clone fragments and a score is given to each clone fragments based on the pre-defined weights. The approach is able to achieve high AUC and emphasize that the code churn, complexity, and the size of the newly-introduced code are highly influential in determining the life of the code clones.

To summarize, code clone genealogy information is used in a different context (e.g., consistency-maintenance, evolutionary patterns, the life of a code pair) in the software project to improve multiple aspects of clone maintenance. We use clone genealogy information to train different machine learning models (learning-to-rank, classification, and regression) that are able to rank the code clones at a commit level.

7 Threats to Validity

In this section, we discuss the threats to the validity of our approach.

Threats to conclusion validity concern the relation between the treatment and the outcome. In our case, threats to conclusion validity concern the errors that occurred when processing the code clones. The accuracy of the clone detection is dependant on the clone detection tool used. To achieve a high accuracy, we use iClones recommended by Savalajeko *et al.* [43] who compare multiple code clone detection tools. We use the same setting as recommended by the results of the comparison.

SZZ is the de facto standard in identifying the fault-inducing commits and used by existing studies [6]. The identification of fault-prone commits and fault-prone clone pairs is performed using the SZZ approach. The approach considers that the fault is introduced before the creation of a bug report, and no commit between the bug report creation and the bug fixing commit is considered. The limitations of the SZZ approach are applicable to our data [21].

Threats to external validity concern the selection of projects and the analysis methods. To mitigate the issue of our results being biased towards a particular set of projects, we use well-defined selection criteria beforehand to include large-scale open-source software projects. The projects are selected from different domains of software development. The clone detection is varied (i.e., the number of clones identified for each project is different) among the selected projects. We aim to show the diversity of the selected projects to ensure that our results are not biased towards a specific language. We include projects from two different languages.

We analyze all revisions of the projects from their creation date until September 2020. The code clones detected, issue reports, and features calculated are valid for this period. A selection of projects from a different time period can result in a different number of clones detected, different clone genealogies, and different values for the features identified.

Prior studies [27] [51] indicate that it is essential to identify different aliases used by developers in an open-source project. We fix the problem of disambiguation of identity (due to multiple aliases) as follows. Instead of the previous approaches that are valid for mailing lists and other similar datasets, we use the GitHub API to retrieve the GitHub account information of the committers. Each commit has an associated email for the committer; we are able to verify that every committer has a different GitHub account. Our approach is similar to the prior studies solving the disambiguation problem [2]. However, similar to existing approaches, we are unable to identify whether a developer has multiple GitHub accounts.

Threats to internal validity concern the possibility of generalizing our results. We select GitHub because it is the most popular platform for open-source projects, and in addition, commits, pull requests, and issue reports are readily available. We use iClones as it can achieve higher accuracy in detecting the code clones. The projects from Java and C programming languages are popular in open-source. Our study can be extended to software projects from other programming languages hosted on different platforms, and other clone detection tools can be used to detect the clones in the projects.

8 Conclusion

Previous studies report that code clones should be adequately maintained to reduce maintenance costs and prevent future faults. The occurrence of similar code fragments in the software project can be harmful, leading to the introduction of bugs in the software projects. In this paper, we examine the possibility of ranking clone fragments based on the risk of future fault occurrences. Specifically, we identify code clones from 534,672 commits from 34 Java and 18 C open-source software projects from GitHub. We identify 469,239 clone genealogies from the studied projects and examined the bug association to the clone genealogies. We then calculate 28 different features related to process, product, genealogy, and users for the clone pairs identified. We experiment with different learning-to-rank (LtR), classification, and regression machine learning models. Our findings can be described as follows.

– We train LtR models to identify the effectiveness of LtR algorithms on our dataset. Our results show that LightGBM achieves a precision of 0.72 to rank the code clones for fault-proneness.
– We use classification approaches to predict the probability of a code clone having fault or not. Random Forest achieves the highest AUC (0.96) among the studied classification approaches.
– We use regression approaches to predict the number of faulty changes of the code clones. Our 10-fold cross-validation on six different regression approaches shows promising results and indicates that LightGBM (0.87 R-squared) is useful in predicting the number of faulty changes of a code clone.
– We provide specialized models using early and mature phase data of the projects. Our analysis shows that as projects become mature, information about the history of the code clones can improve the performance of prediction models of future fault-proneness. Our specialized models can be used by developers for new projects as well as projects that are mature.
– We build a mixed-effect model to identify the significant features for predicting the proportion of faulty changes of the code clones. Our analysis identifies that *UnqUsers* (i.e., the number of unique developers making

changes to a clone pair) and *age_commits* (i.e., the number of commits that a clone pair has changed) has a significant effect on the prediction. Developers can focus on the top significant features for a quick suggestion to prevent faults in the future.

Data Availability Statement

The datasets generated during and/or analysed during the current study are available in the Zenodo repository[7].

9 Appendix

[7] https://zenodo.org/record/7229977

Table 6 Details of the selected projects

Project Name	**Commits**	**Issues**	**SLOC**	**% of files**	**Genealogies**
		Java projects			
Anki-Android	10,647	18,079	402.2k	91.70	3,303
che	8,733	10,474	475.5k	73.80	1,056
checkstyle	9,454	12,108	457.4k	97.80	7,705
druid	10,496	8,878	1.2m	94.50	61,718
elasticsearch	53,815	69,010	3.2m	99.80	4,544
framework	18,969	6,879	867.9k	95.50	11,961
gatk	4,173	7,364	2.2m	93.70	22,651
graylog2-server	16,934	7,498	720.3k	73.90	5,782
grpc-java	4,327	17,698	283.1k	98.30	69,248
jabref	15,271	6,991	1.3m	92.70	4,394
k	15,997	1,704	243.3k	83.50	6,026
k-9	9,579	7,130	177.6k	75.80	2,560
mage	31,307	5,007	1.9m	99.90	15,520
minecraftForge	7,451	5,967	136.1k	99.30	659
molgenis	23,001	6,918	359.8k	86.70	14,891
muikku	16,970	5,156	318.4k	50.20	23,836
nd4j	7,021	10,636	467.0k	99.80	45,413
neo4j	68,916	12,676	837.6k	77.50	60,857
netty	9,910	1,587	476.2k	98.60	13,750
openhab	9,687	9,271	968.5k	99.50	2,678
osmand	64,012	10,091	939.3k	95.70	7,688
pinpoint	11,290	6,451	635.4k	89.30	49,191
presto	17,837	4,987	1.4m	98.80	189
product-apim	7,383	6,451	445.6k	91.60	12,871
realm-java	8,318	6,918	199.9k	83.80	13,540
reddeer	1,550	4,171	136.4k	93.60	20
rxjava	5,762	8,374	474.9k	99.90	8,866
smarthome	5,162	7,099	514.0k	93.80	1,348
spring-boot	27,850	25,295	625.5k	98.80	7,841
terasology	10,405	3,043	321.2k	97.80	56,837
wildfly-camel	1,662	6,216	141.3k	99.20	741
xchange	10,434	2,094	655.6k	100.00	3,545
xp	22,615	3,750	423.0k	95.10	68
zaproxy	7,450	9,271	516.4k	72.00	330
		C projects			
betaflight	16,458	6,467	3.2m	94.50	4,658
cleanflight	16,589	4,554	3m	94.50	2,587
collectd	11,650	6,832	251k	79.50	1,586
fontForge	19,314	3,834	2.8m	97.90	33,848
freeRDP	14,657	8,444	561k	83.00	11,398
inav	10,339	15,916	2.8m	95.60	4,826
johnTheRipper	15,785	11,411	1.1m	75.90	13,874
libgit2	13,602	3,630	413k	98.20	2,657
lxc	9,748	8,464	138k	86.30	9,356
micropython	11,902	4,587	594k	87.00	3,274
mpv	48,889	5,761	235k	88.30	1,952
netdata	11,823	17,907	603k	73.80	23,874
ompi	31,167	10,463	857k	79.90	6,874
radare2	25,047	10,566	1.3m	93.30	8,947
redis	9,867	6,754	303k	82.60	814
riot	33,594	18,392	2.5m	88.30	567
systemd	48,352	3,116	1.5m	88.30	40,785
zfs	6,459	8,375	808k	75.90	328

Table 7 Results of the mixed-effect model for Model$_{all}$, Sorted by χ^2 descendingly

Factor	Coef.	χ^2	Percentage	$Pr(<\chi^2)$	Sign.$^+$	Relationship
(Intercept)	-4.653e^{+00}	1238	0.27	<2.2e^{-16}	***	↘
UnqUsers	1.617e^{-01}	300315	64.53	<2.2e^{-16}	***	↗
age_commits	4.907e^{-01}	117350	25.22	<2.2e^{-16}	***	↗
FCngFreq	-2.470e^{-03}	20924	4.50	<2.2e^{-16}	***	↘
UChgRto	2.668e^{-03}	5917	1.27	<2.2e^{-16}	***	↗
age_days	2.823e^{-04}	4352	0.94	<2.2e^{-16}	***	↗
LvstDis	-7.189e^{-03}	3481	0.75	<2.2e^{-16}	***	↘
CCore	6.768e^{-02}	2295	0.49	<2.2e^{-16}	***	↗
Avg_CT	-8.360e^{-04}	2096	0.45	<2.2e^{-16}	***	↘
sib_cnt	-1.076e^{-02}	1553	0.33	<2.2e^{-16}	***	↘
EFltDens	-5.211e^{-01}	1316	0.28	<2.2e^{-16}	***	↘
SLBurst	5.835e^{-02}	1059	0.23	<2.2e^{-16}	***	↗
experience	3.002e^{-05}	820	0.18	<2.2e^{-16}	***	↗
accepted	2.345e^{-04}	709	0.15	<2.2e^{-16}	***	↗
EConChg	1.594e^{-01}	291	0.06	<2.2e^{-16}	***	↗
rejected	-5.577e^{-04}	273	0.06	<2.2e^{-16}	***	↘
TChurn	6.956e^{-05}	250	0.05	<2.2e^{-16}	***	↗
EIncStChg	9.339e^{-02}	244	0.05	<2.2e^{-16}	***	↗
TPC	-8.662e^{-02}	210	0.05	<2.2e^{-16}	***	↘
EConStChg	-6.570e^{-02}	203	0.04	<2.2e^{-16}	***	↘
CCurSt	5.299e^{-02}	119	0.03	<2.2e^{-16}	***	↗
cloc	9.776e^{-05}	105	0.02	<2.2e^{-16}	***	↗
UFileChg	2.491e^{-04}	88	0.02	<2.2e^{-16}	***	↗
CPathDepth	-4.163e^{-03}	66	0.01	3.734e^{-16}	***	↘
EEvPattern	1.088e^{-02}	45	0.01	1.937e^{-11}	***	↗
NumOfBursts	6.726e^{-03}	33	0.01	8.970e^{-09}	***	↗
EChgTimeInt	-1.025e^{-05}	7	0.00	0.006643	**	↘

$^+$Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Table 8 Results of the mixed-effect model for Model$_{mature}$, Sorted by χ^2 descendingly

Factor	Coef.	χ^2	Percentage	$Pr(<\chi^2)$	Sign.$^+$	Relationship
(Intercept)	$-6.772e^{+00}$	324		$<2.2e^{-16}$	***	$\searrow$
age_commits	$5.230e^{-01}$	11639	52.24	$<2.2e^{-16}$	***	$\nearrow$
UnqUsers	$1.228e^{-01}$	2152	9.66	$<2.2e^{-16}$	***	$\nearrow$
UFileChg	$-1.668e^{-03}$	1318	5.92	$<2.2e^{-16}$	***	$\searrow$
FCngFreq	$-1.185e^{-03}$	1248	5.60	$<2.2e^{-16}$	***	$\searrow$
age_days	$5.290e^{-04}$	1119	5.02	$<2.2e^{-16}$	***	$\nearrow$
CPathDepth	$4.664e^{-02}$	987	4.43	$<2.2e^{-16}$	***	$\nearrow$
LvstDis	$1.182e^{-02}$	969	4.35	$<2.2e^{-16}$	***	$\nearrow$
CCore	$1.679e^{-01}$	718	3.22	$<2.2e^{-16}$	***	$\nearrow$
EFltDens	$-9.565e^{-01}$	345	1.55	$<2.2e^{-16}$	***	$\searrow$
NumOfBursts	$-8.230e^{-02}$	331	1.49	$<2.2e^{-16}$	***	$\searrow$
SLBurst	$1.091e^{-01}$	324	1.45	$<2.2e^{-16}$	***	$\nearrow$
TChurn	$-1.598e^{-03}$	155	0.70	$<2.2e^{-16}$	***	$\searrow$
EIncStChg	$2.624e^{-01}$	128	0.57	$<2.2e^{-16}$	***	$\nearrow$
cloc	$-7.454e^{-04}$	117	0.53	$<2.2e^{-16}$	***	$\searrow$
TPC	$-2.227e^{-01}$	92	0.41	$<2.2e^{-16}$	***	$\searrow$
EConChg	$2.528e^{-01}$	48	0.22	$2.564e^{-12}$	***	$\nearrow$
UChgRto	$7.602e^{-04}$	42	0.19	$7.937e^{-11}$	***	$\nearrow$
sib_cnt	$-4.957e^{-03}$	41	0.18	$1.279e^{-10}$	***	$\searrow$
experience	$1.944e^{-05}$	33	0.15	$7.821e^{-09}$	***	$\nearrow$
EEvPattern	$-3.791e^{-02}$	32	0.14	$9.858e^{-09}$	***	$\searrow$
CCurSt	$1.097e^{-01}$	31	0.14	$2.399e^{-08}$	***	$\nearrow$
rejected	$3.149e^{-04}$	24	0.11	$5.928e^{-07}$	***	$\nearrow$
accepted	$7.361e^{-05}$	21	0.09	$4.022e^{-06}$	***	$\nearrow$
Avg_CT	$2.603e^{-04}$	17	0.08	$2.806e^{-05}$	***	$\nearrow$
EConStChg	$7.308e^{-02}$	15	0.07	$6.575e^{-05}$	***	$\nearrow$
EChgTimeInt	$-4.249e^{-05}$	9	0.04	0.001636	**	$\searrow$

$^+$Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Table 9 Results of the mixed-effect model for Model$_{early}$, Sorted by χ^2 descendingly

Factor	Coef.	χ^2	Percentage	$Pr(<\chi^2)$	Sign.$^+$	Relationship
(Intercept)	$-4.400e^{+00}$	103	0.13	$<2.2e^{-16}$	***	↘
UnqUsers	$1.638e^{-01}$	40938	52.42	$<2.2e^{-16}$	***	↗
age_commits	$4.420e^{-01}$	26567	34.02	$<2.2e^{-16}$	***	↗
CPathDepth	$1.082e^{-01}$	3762	4.82	$<2.2e^{-16}$	***	↗
cloc	$5.170e^{-03}$	3736	4.78	$<2.2e^{-16}$	***	↗
SLBurst	$-1.163e^{-01}$	430	0.55	$<2.2e^{-16}$	***	↘
UChgRto	$1.463e^{-03}$	404	0.52	$<2.2e^{-16}$	***	↗
experience	$1.075e^{-04}$	290	0.37	$<2.2e^{-16}$	***	↗
LvstDis	$6.167e^{-03}$	282	0.36	$<2.2e^{-16}$	***	↗
NumOfBursts	$-8.708e^{-02}$	262	0.34	$<2.2e^{-16}$	***	↘
Avg_CT	$-5.338e^{-04}$	198	0.25	$<2.2e^{-16}$	***	↘
CCore	$6.091e^{-02}$	186	0.24	$<2.2e^{-16}$	***	↗
FCngFreq	$-1.450e^{-03}$	138	0.18	$<2.2e^{-16}$	***	↘
EConStChg	$2.030e^{-01}$	129	0.17	$<2.2e^{-16}$	***	↗
EFltDens	$4.681e^{-01}$	127	0.16	$<2.2e^{-16}$	***	↗
EChgTimeInt	$4.041e^{-04}$	125	0.16	$<2.2e^{-16}$	***	↗
age_days	$9.855e^{-05}$	123	0.16	$<2.2e^{-16}$	***	↗
sib_cnt	$5.250e^{-03}$	81	0.10	$<2.2e^{-16}$	***	↗
EConChg	$-3.142e^{-01}$	77	0.10	$<2.2e^{-16}$	***	↘
TChurn	$-2.313e^{-04}$	64	0.08	$8.367e^{-16}$	***	↘
TPC	$1.335e^{-01}$	38	0.05	$7.029e^{-10}$	***	↗
CCurSt	$-7.345e^{-02}$	13	0.02	0.000211	***	↘
EIncStChg	$-7.046e^{-02}$	10	0.01	0.001138	**	↘
rejected	$-1.914e^{-03}$	7	0.01	0.005693	**	↘
EEvPattern	$1.643e^{-02}$	6	0.01	0.013103	*	↗
accepted	$1.771e^{-04}$	2	0.00	0.088527	.	↗
UFileChg	$-5.907e^{-04}$	0	-	0.562663		↘

$^+$Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

References

1. Asaduzzaman, M., Roy, C.K., Schneider, K.A.: Viscad: flexible code clone analysis support for nicad. In: Proceedings of the 5th International Workshop on Software Clones, pp. 77–78 (2011)
2. Avelino, G., Constantinou, E., Valente, M.T., Serebrenik, A.: On the abandonment and survival of open source projects: An empirical investigation. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–12. IEEE (2019)
3. Barbour, L., An, L., Khomh, F., Zou, Y., Wang, S.: An investigation of the fault-proneness of clone evolutionary patterns. Software Quality Journal **26**(4), 1187–1222 (2018)
4. Barbour, L., Khomh, F., Zou, Y.: Late propagation in software clones. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp. 273–282. IEEE (2011)
5. Bates, D., Maechler, M., Bolker, B., Walker, S., Christensen, R.H.B., Singmann, H., Dai, B., Scheipl, F., Grothendieck, G.: Package 'lme4'
6. Berg, K., Svensson, O.: Szz unleashed: Bug prediction on the jenkins core repository (open source implementations of bug prediction tools on commit level). LU-CS-EX 2018-04 (2018)
7. Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, pp. 785–794 (2016)
8. Cohen, J., Cohen, P., West, S.G., Aiken, L.S.: Applied multiple regression/correlation analysis for the behavioral sciences. Routledge (2013)
9. Cordy, J.R., Roy, C.K.: The nicad clone detector. In: 2011 IEEE 19th International Conference on Program Comprehension, pp. 219–220. IEEE (2011)
10. C.S., C.: whatthepatch - python's third party patch parsing library. Online (Accessed August 17th, 2020)
11. Dhaliwal, S.S., Nahid, A.A., Abbas, R.: Effective intrusion detection system using xgboost. Information **9**(7), 149 (2018)
12. Ehsan, O., Hassan, S., Mezouar, M.E., Zou, Y.: An empirical study of developer discussions in the gitter platform. ACM Transactions on Software Engineering and Methodology (TOSEM) **30**(1), 1–39 (2020)
13. Fischer, M., Pinzger, M., Gall, H.: Populating a release history database from version control and bug tracking systems. In: International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings., pp. 23–32. IEEE (2003)
14. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
15. Fox, J., Weisberg, S., Adler, D., Bates, D., Baud-Bovy, G., Ellison, S., Firth, D., Friendly, M., Gorjanc, G., Graves, S., et al.: Package 'car'. Vienna: R Foundation for Statistical Computing (2012)
16. Garg, R., Tekchandani, R.: An approach to rank code clones for efficient clone management. In: 2014 International Conference on Advances in Electronics Computers and Communications, pp. 1–5. IEEE (2014)
17. Göde, N., Koschke, R.: Incremental clone detection. In: 2009 13th european conference on software maintenance and reengineering, pp. 219–228. IEEE (2009)
18. Goutte, C., Gaussier, E.: A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In: European conference on information retrieval, pp. 345–359. Springer (2005)
19. Hanley, J.A., McNeil, B.J.: The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology **143**(1), 29–36 (1982)
20. Hassan, S., Tantithamthavorn, C., Bezemer, C.P., Hassan, A.E.: Studying the dialogue between users and developers of free apps in the google play store. Empirical Software Engineering **23**(3), 1275–1312 (2018)
21. Herbold, S., Trautsch, A., Trautsch, F., Ledel, B.: Issues with szz: An empirical assessment of the state of practice of defect prediction data collection. arXiv preprint arXiv:1911.08938 (2019)

22. Jr, F.E.H.: Harrell miscellaneous. `https://cran.r-project.org/web/packages/Hmisc/Hmisc.pdf`. (Last accessed: August 2019)
23. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: 2009 IEEE 31st International Conference on Software Engineering, pp. 485–495. IEEE (2009)
24. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **28**(7), 654–670 (2002)
25. Kapser, C.J., Godfrey, M.W.: “cloning considered harmful” considered harmful: patterns of cloning in software. Empirical Software Engineering **13**(6), 645–692 (2008)
26. Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., Liu, T.Y.: Lightgbm: A highly efficient gradient boosting decision tree. In: Advances in neural information processing systems, pp. 3146–3154 (2017)
27. Kouters, E., Vasilescu, B., Serebrenik, A., Van Den Brand, M.G.: Who’s who in gnome: Using lsa to merge software repository identities. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 592–595. IEEE (2012)
28. Lafontaine, F., White, K.J.: Obtaining any wald statistic you want. Economics Letters **21**(1), 35–40 (1986)
29. Li, H.: Learning to rank for information retrieval and natural language processing. Synthesis Lectures on Human Language Technologies **7**(3), 1–121 (2014)
30. Li, J., Ernst, M.D.: Cbcd: Cloned buggy code detector. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 310–320. IEEE (2012)
31. Mondal, M., Roy, C.K., Schneider, K.A.: Bug propagation through code cloning: An empirical study. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 227–237. IEEE (2017)
32. Mondal, M., Roy, C.K., Schneider, K.A.: Does cloned code increase maintenance effort? In: 2017 IEEE 11th International Workshop on Software Clones (IWSC), pp. 1–7. IEEE (2017)
33. Nakakoji, K., Yamamoto, Y., Nishinaka, Y., Kishida, K., Ye, Y.: Evolution patterns of open-source software systems and communities. In: Proceedings of the international workshop on Principles of software evolution, pp. 76–85 (2002)
34. Pan, Q., Tang, W., Yao, S.: The application of lightgbm in microsoft malware detection. In: Journal of Physics: Conference Series, vol. 1684, p. 012041. IOP Publishing (2020)
35. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
36. Roy, C.K.: Detection and analysis of near-miss software clones. In: 2009 IEEE International Conference on Software Maintenance, pp. 447–450. IEEE (2009)
37. Saha, R.K., Roy, C.K., Schneider, K.A.: An automatic framework for extracting and classifying near-miss clone genealogies. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp. 293–302. IEEE (2011)
38. Saha, R.K., Roy, C.K., Schneider, K.A., Perry, D.E.: Understanding the evolution of type-3 clones: an exploratory study. In: Proceedings of the 10th Working Conference on Mining Software Repositories, pp. 139–148. IEEE Press (2013)
39. Schwarz, N., Lungu, M., Robbes, R.: On how often code is cloned across repositories. In: Proceedings of the 34th International Conference on Software Engineering, pp. 1289–1292. IEEE Press (2012)
40. Shepperd, M., Bowes, D., Hall, T.: Researcher bias: The use of machine learning in software defect prediction. IEEE Transactions on Software Engineering **40**(6), 603–616 (2014)
41. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? ACM sigsoft software engineering notes **30**(4), 1–5 (2005)
42. Snijders, T.A., Bosker, R.J., et al.: An introduction to basic and advanced multilevel modeling. Sage, London. WONG, GY, y MASON, WM (1985): The Hierarchical Logistic Regression. Model for Multilevel Analysis, Journal of the American Statistical Association **80**(5), 13–524 (1999)

43. Svajlenko, J., Roy, C.K.: Evaluating modern clone detection tools. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 321–330. IEEE (2014)
44. Svajlenko, J., Roy, C.K.: The mutation and injection framework: evaluating clone detection tools with mutation analysis. IEEE Transactions on Software Engineering **47**(5), 1060–1087 (2019)
45. Tang, C., Luktarhan, N., Zhao, Y.: An efficient intrusion detection method based on lightgbm and autoencoder. Symmetry **12**(9), 1458 (2020)
46. Thongtanunam, P., Shang, W., Hassan, A.E.: Will this clone be short-lived? towards a better understanding of the characteristics of short-lived clones. Empirical Software Engineering **24**(2), 937–972 (2019)
47. Walthers, J.: Learning to rank for cross-device identification. In: 2015 IEEE International Conference on Data Mining Workshop (ICDMW), pp. 1710–1712. IEEE (2015)
48. Wang, S., Chen, T.H., Hassan, A.E.: Understanding the factors for fast answers in technical q&a websites. Empirical Software Engineering **23**(3), 1552–1593 (2018)
49. Wang, S., Zou, Y., Ng, J., Ng, T.: Context-aware service input ranking by learning from historical information. IEEE Transactions on Services Computing (2017)
50. Weisberg, S.: Applied linear regression, vol. 528. John Wiley & Sons (2005)
51. Wiese, I.S., da Silva, J.T., Steinmacher, I., Treude, C., Gerosa, M.A.: Who is who in the mailing list? comparing six disambiguation heuristics to identify multiple addresses of a participant. In: 2016 IEEE international conference on software maintenance and evolution (ICSME), pp. 345–355. IEEE (2016)
52. Xie, S., Khomh, F., Zou, Y.: An empirical study of the fault-proneness of clone mutation and clone migration. In: 2013 10th Working Conference on Mining Software Repositories (MSR), pp. 149–158. IEEE (2013)
53. Yang, B., He, Y., Liu, H., Chen, Y., Jin, Z.: A lightweight fault localization approach based on xgboost. In: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), pp. 168–179. IEEE (2020)
54. Yang, X., Tang, K., Yao, X.: A learning-to-rank approach to software defect prediction. IEEE Transactions on Reliability **64**(1), 234–246 (2014)
55. Zhang, F., Khoo, S.c., Su, X.: Predicting change consistency in a clone group. Journal of Systems and Software **134**, 105–119 (2017)
56. Zhao, G., da Costa, D.A., Zou, Y.: Improving the pull requests review process using learning-to-rank algorithms. Empirical Software Engineering **24**(4), 2140–2170 (2019)
57. Zhou, J., Zhang, H.: Learning to rank duplicate bug reports. In: Proceedings of the 21st ACM international conference on Information and knowledge management, pp. 852–861 (2012)