

Automatic Prediction of Developers' Resolutions for Software Merge Conflicts

Waad Aldndni^a, Na Meng^a, Francisco Servant^{b,*}

^aVirginia Polytechnic Institute and State University, Blacksburg, VA, 24060, U.S.A.

^bITIS Software, Universidad de Málaga, Málaga, 29071, Spain

Abstract

In collaborative software development, developers simultaneously work in parallel on different branches that they merge periodically. When edits from different branches textually overlap, conflicts may occur. Manually resolving conflicts can be tedious and error-prone. Researchers proposed tool support for conflict resolution, but these tools barely consider developers' preferences. Conflicts can be resolved by: keeping the local version only (KL), keeping the remote version only (KR), or manually editing them (ME). Recent studies show that developers resolved the majority of textual conflicts by KL or KR. Thus, we created a machine learning-based approach RPREDICTOR to predict developers' resolution strategy (KL, KR, or ME) given a merge conflict.

We did large-scale experiments on the historical resolution of 74,861 conflicts. Our experiments show that RPREDICTOR achieved 63% F-score for within-project prediction and 46% F-score for cross-project prediction. Compared with other classifiers, RPREDICTOR provides the highest effectiveness when using a random forest (RF) classifier. Finally, we proposed a variant technique RPREDICTOR_v, which enables developers to customize its prediction conservativeness. For a highly conservative setting, RPREDICTOR_v achieved 34% effort saving while minimizing the risk of producing incorrect prediction labels.

Keywords: Software merge, textual conflicts, conflict resolution, prediction

1. Introduction

In collaborative software development, programmers often create separate branches to perform distinct maintenance tasks (e.g., add new features, fix bugs, or refactor code) in parallel. When developers merge edits from different

*Corresponding author

Email addresses: waada@vt.edu (Waad Aldndni), nm8247@vt.edu (Na Meng), fservant@uma.es (Francisco Servant)

5 branches, separate edits that were simultaneously applied to the same line of code can conflict with each other.

1.1. Background

Manual resolution of such conflicts is usually challenging and time-consuming. A prior study [49] shows that 56% of developers deferred resolving a merge conflict due to various reasons (*e.g.*, the complexity, large size, or big number of locations of conflicting code). In the period of time between conflicts occur and they get resolved, conflicts can grow and become more difficult to resolve [49]. Vale *et al.* [68] identified factors that make conflicts hard to solve, including the number of conflicting lines of code, the number of conflicting chunks, the number of lines of code changed, and the number of files changed. By conducting surveys with developers, Costa *et al.* [21] showed that the developer performing a merge might not fully understand the changed code or the rationale behind the change, or may not have the expertise to determine the impact of the change. Nelson *et al.* [49] interviewed 10 software developers, and revealed that developers need better tools to facilitate the understanding and resolution of merge conflicts. All these studies motivated us to explore new ways of automatic conflict resolution.

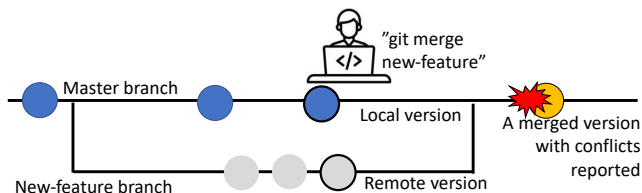


Figure 1: Developers use textual merge (*e.g.*, `git-merge`) to merge branches and reveal conflicts

As illustrated in Figure 1, developers typically adopt text-based tools (*e.g.*, `git-merge` [1]) to tentatively merge the latest version of their own branch (*i.e.*, **local version (L)**) with the latest version of a specified branch (*i.e.*, **remote version (R)**), and to detect textual conflicts in this process. Because such tools treat programs as plain text, they can merge the code in ways that are syntactically or semantically incorrect, due to code mismatches between branches [19, 50, 64]. To improve over textual merge, researchers proposed tools that analyze the syntactic structures of programs, to better detect and resolve conflicts [13, 12, 71, 65]. For instance, JDime [12] matches Java code based on abstract syntax trees (ASTs). It conducts tree-based merge instead of text-based merge for each matching node pair, to better align code and integrate as many edits as possible between branches.

Existing tools resolve conflicts by integrating edits with the best effort, but they overlook the preferences of developers. Two recent studies show that when resolving conflicts, instead of merging edits from both branches, developers often keep edits from the local (L) or remote (R) version [69, 27]. Specifically, Yuzuki *et al.* [69] examined 779 Java merge conflicts, and found that developers resolved 99% of conflicting methods by keeping only one of the conflicting versions. Similarly, Ghiotto *et al.* [27] inspected 616 textual conflicts; they observed that developers resolved 56% of cases by keeping only the L or R version. As the

studies were done by different researchers on distinct datasets and the adopted granularity (method vs. line) varies, the reported percentages are different.

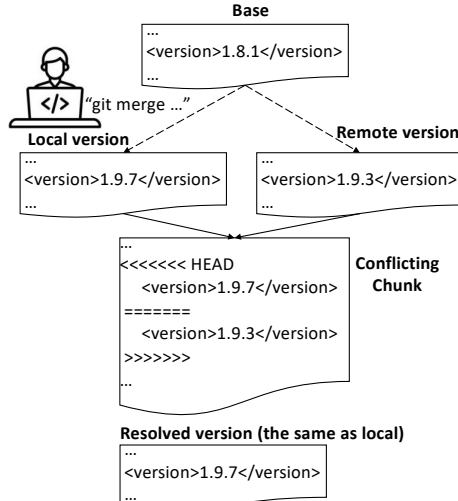


Figure 2: Versions related to a merge conflict

Figure 2 shows an exemplar conflict manually resolved by developers. For this example, L and R simultaneously updated the version number of a library
 45 dependency in distinct ways. As a result, the text-based merge (*e.g.*, git-merge) reveals a conflicting chunk, which uses the format “<<<<<< HEAD
 ... ===== ... >>>>>>” to mark the conflicting edits between versions. To resolve the conflict, developers simply picked the edit from one version (*i.e.*,
 50 L) instead of trying to combine the branch edits somehow.

1.2. Motivation

Generally speaking, developers resolve conflicts via three main strategies: choosing the local version while discarding the remote one (**KL**), choosing the
 remote version while discarding local (**KR**), or modifying edits from either or
 55 both branches for edit integration (**ME**). Inspired by the two studies mentioned above, we wanted to create a novel approach that resolves merge conflicts by considering developers’ preferences. As our new approach predicts the resolution strategy for any given conflict, we expect it to help developers in two ways. First, when it correctly predicts the KL or KR strategy, the approach can auto-
 60 matically apply the strategy and resolve the conflict. This will save developers time and manual effort, which would have been spent on understanding and resolving that conflict. The effort savings provided by this automatic prediction are potentially very high, since past evidence shows that the majority of conflicts get resolved by KL and KR [69, 27]. Second, when our approach predicts
 65 the ME strategy, it reminds developers to carefully inspect the local and remote branches, in order to cautiously handle the given conflict.

1.3. Our Research

To explore the feasibility of creating a predictor for conflict-resolution strategies, we first did an empirical study to characterize the conflicts in software version history that get resolved with different strategies. We gathered 15,758 conflicts from 100 open-source software repositories, and studied 12 features to characterize each conflict from different perspectives. Our statistical analysis shows a strong correlation between the resolution decisions of developers and all features, indicating a strong potential for successfully building a resolution predictor.

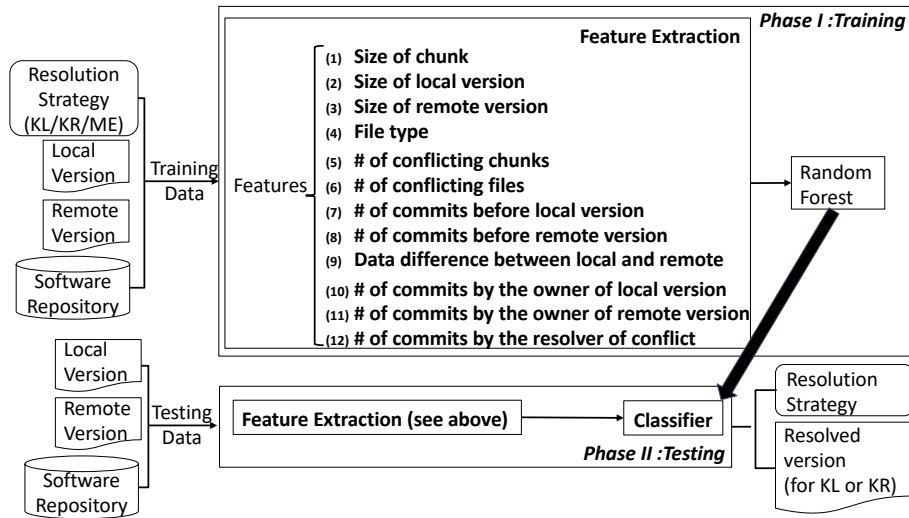


Figure 3: RPRELECTOR has two phases: training and testing

Leveraging the 12 features revealed by our study, we designed and implemented an approach—RPRELECTOR—to automatically predict resolution strategies. As shown in Figure 3, RPRELECTOR has two phases: training and testing. In Phase I, RPRELECTOR extracts features for each conflict in a set of merge conflicts that were already resolved in the past, and trains a three-class random forest (RF) classifier. In Phase II, RPRELECTOR takes in any new conflict together with the software repository holding that conflict, extracts features, and applies the trained classifier to recommend a strategy. When the strategy is KL or KR, RPRELECTOR also outputs the resolved version.

To evaluate RPRELECTOR, we conducted large-scale experiments with 74,861 conflicts extracted from the version history of 482 open-source projects. We applied RPRELECTOR to perform both within-project and cross-project prediction tasks. For the within-project setting, in each repository, we used the oldest 90% of resolved conflicts to train RPRELECTOR and the remaining 10% of resolved conflicts for testing. RPRELECTOR predicted resolutions with 63% F-score. For the cross-project setting, we performed 10-fold cross validation. Namely, we divided the 482 software repositories evenly into 10 folds. In each experiment, we

leveraged the conflict data in nine folds for training and used the conflict data from the remaining fold for testing. We repeated the experiment 10 times, with
95 each experiment using a different fold for testing. RPREDICTOR recommended resolutions with 46% F-score.

We also evaluated the sensitivity of RPREDICTOR to different amounts or ages of training data, and to different machine learning (ML) algorithms. We found that as more training data is provided, RPREDICTOR’s effectiveness either
100 increases or stabilizes; nevertheless, it does not change consistently with the age of training data. Compared with other ML algorithms, random forest leads to the best effectiveness of RPREDICTOR. Finally, we designed a customizable variant, RPREDICTOR_v, which allows developers to customize how conservatively they want RPREDICTOR_v to make its predictions, *i.e.*, how inclined it should be
105 to predict the ME resolution, to reduce the ratio of incorrectly predicted KL or KR. For a highly conservative setting (94% C-score), RPREDICTOR_v achieved 34% effort savings; making RPREDICTOR_v less conservative but keeping its conservativeness score (C-score) over 80%, we got RPREDICTOR_v to achieve up to 64% effort savings. We made the following contributions in this paper:

- 110 • A novel empirical study of 12 characteristics of 15,758 conflicts, to understand their correlation with resolutions KL, KR, or ME.
- A novel tool RPREDICTOR, that leverages machine learning (ML) to predict the resolution strategy for a given conflict.
- A comprehensive evaluation to assess the prediction effectiveness of RPREDICTOR, with 74,861 conflicts from 482 Java open-source repositories.
115
- An evaluation of RPREDICTOR’s sensitivity to different configurations related to ML, including (1) the ratio of training and testing data, (2) the balanced or unbalanced data distribution among resolution strategies, (3) the age of training data, and (4) the choice of learning algorithms.
- 120 • A customizable variant RPREDICTOR_v, which enables developers to choose more or less conservative results at the trade-off of lower or higher effort savings, respectively.

2. Dataset Construction

Ghiotto *et al.* [27] recently conducted an empirical study on merge conflicts, and created a dataset of conflicts from 2,731 GitHub repositories. To study
125 developers’ preferences on conflict resolution and to explore new approaches of resolution prediction, we decided to create our datasets based on their data because of its comprehensiveness and representativeness.

To create the dataset, Ghiotto et al. first used the GitHub API to select
130 1,997,541 projects. Then they collected information about each project such as the last update date, the size of its development team, and the code size. Next, they selected all Java projects that have at least one commit during January

2015 and March 2016. A project is considered a Java project if the percentage of source code written in Java is greater than that of code written in any of the other languages. Finally, they discarded the projects that were forks of other projects in the dataset or had no conflict reported by git-merge in Java files for any merge commits. This led to 2,731 projects with 175,805 conflicting chunks.

For our study, we downloaded Ghitto’s dataset and refined it by taking two steps. First, we removed the projects whose developers resolved conflicts by taking only one or two major strategies e.g., jsoup [7] and platform_frameworks_base [4]. Namely, if a project (1) has at least 50% of conflicts resolved via a single strategy (KL, KR, or ME) or (2) never uses a certain strategy (e.g., KL), we remove the project. In this way, we ensured that each of the remaining repositories had a relatively balanced distribution of conflicts among KL, KR, and ME. After this step, 609 projects remained in our dataset (e.g., XCoLab [5] and jgralab [6]). Second, we removed the projects whose codebases were no longer available on GitHub, and our final corpus became 582 projects. Table 1 shows some characteristics of the 582 software repositories. As shown in the table, each project involves at least 2 developers and at most 426 developers, with the mean value 25 and standard deviation 43. Each repository has at least 24 commits, and at most 190,851 commits. In each repository, there are 2–22,020 merging scenarios, while the number of conflicting chunks varies in 5–5,114. All these numbers imply that the software projects are not toy examples; many of them are large or complex projects involving many developers and having long version histories.

Table 1: Characteristics of the 582 software repositories included by our dataset

	Min	Max	Mean	Standard Deviation
Number of Developers	2	426	25	43
Number of Commits	24	190,851	2,820	9,022
Number of Merges	2	22,020	323	1,045
Number of Chunks	5	5,114	156	368

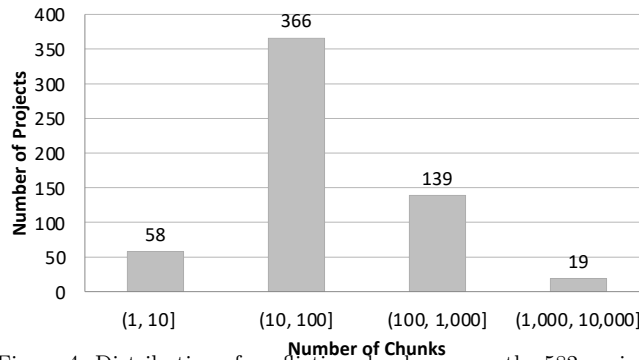


Figure 4: Distribution of conflicting chunks among the 582 projects

Moreover, we analyzed the distribution of conflicting chunks among projects. As shown in Figure 4, 58 out of the 582 projects have 2–10 conflicting chunks in version history; 366 projects have 11–100 chunks; 139 have 101–1,000 chunks; and 19 projects have over 1,000 chunks. Such a distribution implies that some projects contribute a lot more chunks than the others, and may bias our exper-

160 iment results. However, in total the dataset includes 90,619 conflicting chunks, while the largest number of conflicting chunks contained by any single project is 5,114. It means that there is no project dominating the whole dataset, so the impact of any potential bias is limited.

165 As shown in Table 2, after refining the original dataset of Ghitto *et al.*, we obtained 582 software repositories. Among the 90,619 conflicts contained by these repositories, there are 32,065, 24,423, 34,131 conflicts separately resolved via KL, KR, and ME.

170 We randomly sampled 100 repositories in the 582 repositories, to conduct a characterization study of conflicts (see Section 3). This sample set includes 15,758 conflicts, among which 5,519 conflicts were resolved via KL, 4,357 conflicts were resolved via KR, and 5,882 conflicts were resolved via ME. Based on the characterization study, we created RPREDICTOR, and evaluated the tool using all data from the remaining 482 repositories. By making observations on a subset of data and assessing new approaches on the remaining data, we can
175 examine whether the insights gained from some data are generalizable to other unseen data.

Table 2: The datasets used in our research

	# of Repositories	# of Conflicts resolved by			
		KL	KR	ME	Total
Data used in our characterization study	100	5,519	4,357	5,882	15,758
Data used in the tool evaluation	482	26,546	20,066	28,249	74,861
Total	582	32,065	24,423	34,131	90,619

3. Our Characterization Study

180 We characterized all resolved conflicts in the randomly sampled 100 repositories by defining and measuring 12 features. We defined these features based on the insights we learnt from prior studies [27, 49, 16, 68], concerning factors that may impact developers’ decisions on conflict resolution. We organized the features into four categories: (C1) content of the merge conflict, (C2) the scenario in which the conflict happened, (C3) software evolution that led to the conflict, and (C4) experience of the developer(s) involved in the conflict. We
185 describe them in more detail below.

C1. Conflict Content: We hypothesize that developers often observe the conflict content when they try to resolve a conflict [27, 49, 16, 68]. We defined four features to characterize the content of a conflicting chunk:

190 **F1. Size of Chunk** counts the lines of code (LOC) contained by any given conflicting chunk.

F2. Size of Local Version counts the LOC between “<<<<<< HEAD” and “====”. Namely, for each conflicting chunk, it counts the unique code coming from the local version.

195 F3. **Size of Remote Version** counts the LOC between “====” and “>>>>>>”.
Namely, for each chunk, it counts the unique code derived from remote.

F4. **File Type** reflects the type of the file containing the conflicting chunk.
Different resolutions may be popular in different types of files.

200 Notice that $F1 > F2 + F3$, because a conflicting chunk consists of (1) the unique
code from L and R and (2) some common code (*e.g.*, program context) shared
between versions. We believe that when developers resolve merge conflicts, the
surrounding context is important for them to decide (i) which branch edits fit
better and (ii) how to integrate branch edits into the context. Thus, both the
conflicting edits and surrounding context can influence developers’ resolution
strategies, and we included F1–F3 into our study.

205 **C2. Merging Scenarios:** The complexity of a merging scenario (*i.e.*, the
scenario where git-merge is applied to merge two branch versions) could make
developers defer their responses to conflicts [27, 49, 16, 68]. We defined two
features to capture the complexity:

210 F5. **Number of Conflicting Chunks** counts the conflicting chunks reported
by git-merge for a merging scenario .

F6. **Number of Conflicting Files** counts the number of conflicting files in
a merging scenario.

215 **C3. Evolution of Changes:** It is possible that for a given conflict, how
local and remote versions separately evolved can influence developers’ resolution
strategies [49, 16]. We hypothesize that branches with longer history are less
likely to be discarded, and defined the following three features accordingly:

F7. **Number of Commits before Local** counts the commits or versions
standing between the base and local versions, on the branch where the
local version resides.

220 F8. **Number of Commits before Remote** counts the commits or versions
standing between the base and remote versions, on the branch where the
remote version resides.

225 F9. **Date Difference between Local and Remote** counts the time interval
(*i.e.*, days) between the check-in dates of local and remote. We hypoth-
esized that an increasing number of days between the check-in dates of
local and remote versions can make a conflict harder to solve, and thus
may influence developers’ decisions for its resolution.

230 **C4. Developer Experience:** The experience of developers can considerably
impact how they understand and resolve conflicts [49, 16, 68]. We hypothesize
that the number of historical commits checked in by a developer can reflect
his/her experience with the software project. We extracted the user IDs of
developers, and defined the following three features:

235 F10. **Number of Commits by The Owner of Local:** If a developer
checked in the local commit for the current merging scenario, we consider
that developer as the owner of local. While multiple developers might
contribute changes to the local branch, we assign the ownership of local
version to the last committer. This is because committers often review all
existing code (including other developers' edits) and their modifications
240 before committing changes. This feature counts the commits checked in
by the owner of local, before that developer committed the local version.

245 F11. **Number of Commits by The Owner of Remote:** If a developer
checked in the remote commit for the current merging scenario, we con-
sider that developer as the owner of remote. When multiple developers
contribute changes to the remote branch, we assign the ownership of re-
mote version to the last committer. This is because the last committer
typically reviews all existing code and his/her own changes before check-
ing in the commit. This feature counts the commits checked in by the
owner of remote, before that developer committed the remote version.

250 F12. **Number of Commits by The Resolver of Conflict:** If a developer
checked in the merging commit with conflict resolution for the current
merging scenario, we consider that developer as the resolver of conflict.
We believe that the resolution strategies vary with resolvers. In reality,
to predict developers' resolution strategy for a given conflict, it is hard
to know beforehand who will resolve the conflict. However, it is still use-
255 ful to explore the prediction power of this feature, because the potential
predictors-to-build can take in manually entered resolver's user ID to pre-
dict the resolution strategy for a specified conflict. This feature counts the
commits checked in by the developer who resolved a given conflict, before
that conflict resolution.

260 To study whether these 12 features impact developers' resolution strategies, we
applied statistical analysis to compare the values of these features for conflicts
separately resolved by KL, KR, and ME. As mentioned in Section 2, in this
study, we used in total 15,758 conflicts from 100 randomly sampled repositories.

3.1. Statistical Analysis via H Test

265 Among the 12 features mentioned above, there are 11 features (except F4)
that have numeric values. For each of these features F_i ($i \in [1, 12], i \neq 4$), we
measured its value for each conflict. We separated merge conflicts into three
groups, according to the resolution strategies applied to them. We use C_L to
refer to the conflicts resolved by KL, use C_R to refer to the conflicts resolved by
270 KR, and use C_M for those resolved by ME.

To study whether any of these features can be used to predict developers'
resolution strategies, we applied the Kruskal-Wallis H test [46, 43, 8]; it is a
statistical test to decide if three or more groups of samples come from the same
distribution on a variable of interest (*e.g.*, chunk size or number of conflicts). H

275 test is a non-parametric test, as it does not assume a normal data distribution
 (none of our studied features follow a normal distribution). For each group
 of samples, H test sorts data into ascending order, assigns ranks to the sorted
 data points, and thus converts the given values into their ranks. Namely, in the
 conversion process, the smallest value gets a rank of 1, the next smallest gets a
 280 rank of 2, and so on. Among the given three or more sample groups, H test is
 applied to validate the following hypotheses:

- H_0 : The mean ranks of different groups are the same.
- H_1 : The mean ranks of different groups are not the same.

Table 3: The statistical analysis results for F1–F3 and F5–F12

F_i	Mean Ranks			P-value
	C_L	C_R	C_M	
F1. Size of Chunk	25	26	72	0.000046
F2. Size of Local Version	11	13	35	0.000093
F3. Size of Remote Version	14	12	37	0.000129
F5. Number of Conflicting Chunks	55	61	29	0.000000
F6. Number of Conflicting Files	24	28	16	0.000000
F7. Number of Commits before Local	62	62	43	0.000000
F8. Number of Commits before Remote	91	138	96	0.000000
F9. Date Difference between Local and Remote	5	5	4	0.000055
F10. Number of Commits by The Owner of Local	655	558	603	0.000051
F11. Number of Commits by The Owner of Remote	530	530	548	0.002683
F12. Number of Commits by The Resolver of Conflict	621	540	584	0.010751

Table 3 presents the H test results for all features except F4. For any fea-
 285 ture F_i , a p-value lower than 0.05 implies that the groups (i.e., C_L , C_R , and
 C_M) are from significantly different data distributions, which means that the
 corresponding feature could help predict developers’ resolution strategies. As
 shown in the table, all of the 11 features have p-values lower than 0.05; thus,
 we decided to use these features to train a resolution predictor in Section 4.

290 **Finding 1:** The H test shows that all 11 numeric features (F1–F3 and F5–
 F12) of conflicting chunks can help predict developers’ resolution strategies.

3.2. Statistical Analysis via Chi-Square Test

F4 is different from the other features, because it is a categorical variable
 to characterize file types for conflicts, while the other features are numeric vari-
 ables to count numbers related to a given conflict. To study whether file types
 295 help predict developers’ resolution strategies, we decided to use the chi-square
 test [54]—a statistical test applicable to sets of categorical data, to evaluate how
 possibly any observed difference between the sets happened by chance. Specif-
 ically, in our study, after extracting all file-type information for conflicts, we
 clustered the file types into two big categories: source-code files and non-code
 300 files. We then counted the frequency of each category for each resolution strat-
 egy to obtain a contingency table (see Table 4). Source-code files include files
 written in any programming language, such as Java and Python; non-code files
 include all other kinds of files, such as configuration files and documentation.

Notice that we decided not to use file types as they are to create the contin-
 305 gency table for two reasons. First, we observed 96 file types in the 100 studied

Table 4: Data distribution of conflicts between the two file categories

File Category	# of Files in Each Group			Total
	C_L	C_R	C_M	
Source-Code File	3,930	3,375	4,565	11,870
Non-Code File	1,591	983	1,319	3,893

Java projects. Among those types, Java is the biggest one and covers thousands of conflicts, while many rare file types only cover one or two conflicts. Such an extreme unbalanced conflict distribution among file types can make our statistical analysis useless or even misleading. To ensure the relatively balanced data distribution across categories, we decided to create the 2 big categories out of 96 file types. Second, if we used the file types as they are, our statistical analysis results may be limited to the 96 file types we studied, but not generalize well to larger datasets that have a lot more file types. Clustering raw file types into two big categories helps ensure the generalizability of our study results, because the two big categories remain the same no matter how many more concrete file types are included by larger datasets.

We defined the following hypotheses for our chi-square test:

- H_0 : No association exists between file categories and resolution strategies.
- H_1 : There is association between file categories and resolution strategies.

Our statistical analysis results have chi-square = 77.5874, and $p = 0.0000$. The results imply that file categories are related to developers’ resolution strategies, so we can exploit F4 to train a resolution predictor (Section 4).

Finding 2: *The Chi-square test shows that the file categories of conflicting chunks (F4) can help predict developers’ resolution strategies.*

4. Approach

Our characterization study (see Section 3) shows the feasibility of training a machine-learning model to predict developers’ resolution strategies for conflicts. Therefore, we designed and implemented a new approach RPREDICTOR. As shown in Figure 3, RPREDICTOR has two phases: training and testing. Phase I analyzes the conflicts already resolved by developers to train a three-class classifier. Phase II takes a merge conflict from a software repository, and leverages the trained classifier to predict whether developers will resolve it via KL, KR, or ME. If KL or KR is predicted, in addition to outputting the resolution strategy, RPREDICTOR also outputs the resolved version to automate conflict resolution and thus improve programmer productivity. In both phases, RPREDICTOR extracts 12 features for each conflict. For implementation, we used scikit-learn [55]—a Python machine-learning library to train and test a classifier. The scikit-learn library features various classification, regression, and clustering algorithms. By invoking APIs provided by the library, RPREDICTOR uses random forest (RF) to train its three-class classifier.

Because 11 of the 12 features are numeric variables (i.e., F1-F3 and F5-F12), we provided their numeric values as inputs to RPREDICTOR. One feature

(F4) is categorical, with two category labels as “source code file” and “non-code file”. To provide numeric values to RPREDICTOR for F4, we applied one-hot encoding [29] for category-to-vector conversion. Namely, we used the vector [1, 0] to represent the first category, and used [0, 1] to represent the second.

5. Evaluation

We conducted a variety of experiments to investigate the following seven research questions (RQs):

- **RQ1:** How effectively can RPREDICTOR predict developers’ resolutions in the within-project setting?
- **RQ2:** How effectively can RPREDICTOR predict developers’ resolutions in the cross-project setting?
- **RQ3:** How effectively can RPREDICTOR predict developers’ resolutions given projects with unbalanced distributions of resolution strategies?
- **RQ4:** How sensitive is RPREDICTOR to the amount of training data?
- **RQ5:** How sensitive is RPREDICTOR to the age of training data?
- **RQ6:** How sensitive is RPREDICTOR to the adopted machine-learning algorithm?
- **RQ7:** How sensitive is RPREDICTOR_v to different prediction thresholds?

This section will first introduce our evaluation metrics (Section 5.1), and then present our experiments as well as the results for each research question (Sections 5.2–5.8).

5.1. Evaluation Metrics

In our experiments, we executed our studied techniques to obtain a **prediction** for each one of the merge conflicts in our studied dataset. As **ground truth** for each conflict, we observed the resolution strategy employed by the developer that resolved it in our dataset. We then assessed the effectiveness of a technique by comparing its prediction to the ground truth for each conflict, applying multiple metrics. To facilitate discussion, in this section, we index the three conflict resolution strategies and refer to them as $S_i (i \in [1, 3])$. Namely, S_1 refers to KL (keep the local version); S_2 refers to KR (keep the remote version); S_3 refers to ME (resolution with manual edits). We defined and calculated the following metrics to evaluate effectiveness:

Precision (P_i) measures, among all the conflicts labeled with S_i by a technique, what ratio of them were actually resolved by S_i .

$$P_i = \frac{\# \text{ of conflicts correctly labeled as } "S_i"}{\text{Total } \# \text{ of conflicts labeled as } "S_i"} \quad (1)$$

Recall (R_i) measures, among all conflicts that were resolved by S_i , what ratio of them were labeled by a technique as S_i .

$$R_i = \frac{\# \text{ of conflicts correctly labeled as } "S_i"}{\text{Total } \# \text{ of conflicts that were resolved via } S_i} \quad (2)$$

Both precision and recall vary within [0%, 100%]. The higher, the better.

F-score (F_i) is the harmonic mean of precision and recall. It provides a way to measure a model’s accuracy based on precision and recall. F also varies within [0%, 100%]. The higher value we get, the better.

$$F_i = \frac{2 \times P \times R}{P + R} \quad (3)$$

Aggregated (Overall) metrics (P , R , F): With the above effectiveness metrics computed for each resolution strategy, we further evaluated the overall effectiveness of a technique by computing the *weighted* average among all strategies. Formally, if we use Γ to represent P or R , and use n_i to represent the number of testing samples in S_i , then the overall effectiveness in terms of precision and recall can be computed as

$$\Gamma_{overall} = \frac{\sum_{i=1}^3 \Gamma_i * n_i}{\sum_{i=1}^3 n_i} \quad (4)$$

Finally, the overall F is computed with:

$$F_{overall} = \frac{2 \times P_{overall} \times R_{overall}}{P_{overall} + R_{overall}} \quad (5)$$

Conservativeness Score (C) or C-score: We defined this metric because different prediction mistakes have different consequences. If a conflict resolved by KL or KR is incorrectly predicted as ME, the technique makes a *conservative* mistake: it misses the opportunity of saving developers’ manual effort, but does not mislead developers to blindly take resolution suggestions. However, if a conflict resolved by ME is incorrectly predicted as KL or KR, the technique makes a more serious mistake: it automatically resolves the conflict using a different strategy than what the developer would have preferred, and thus produces an incorrectly merged version. We created a C metric to measure the ratio of predictions that are *conservative*, *i.e.*, that do not cause any incorrect automatic resolution. Conservative predictions include (1) correct predictions, and (2) any conflict resolved via KL or KR but labeled as ME. C scores range within [0%, 100%]; the higher, the better.

$$C = \frac{\# \text{ of conflicts conservatively labeled}}{\text{All predictions}} \quad (6)$$

5.2. RQ1: Effectiveness of Within-Project Prediction

For each software project in our dataset, we leveraged 90% of the oldest resolved conflicts to train RPREDICTOR, and then used the remaining 10% of

Table 5: The prediction counts for RPREDICTOR and Baseline in the within-project setting

Ground Truth	# of conflicts		RPredictor			Baseline		
	Training	Testing	KL	KR	ME	KL	KR	ME
KL	23,610	2,936	1,815	318	803	1,002	977	957
KR	18,087	1,979	343	931	705	678	649	652
ME	25,472	2,777	361	356	2,060	984	892	901
Total	67,169	7,692	2,519	1,605	3,568	2,664	2,518	2,510

Table 6: Effectiveness measurements for within-project prediction

	# of conflicts		RPredictor				Baseline			
	Training	Testing	P	R	F	C	P	R	F	C
KL	23,610	2,936	72%	62%	67%	-	37%	34%	35%	-
KR	18,087	1,979	58%	47%	52%	-	25%	32%	28%	-
ME	25,472	2,777	58%	74%	65%	-	35%	32%	34%	-
Overall	67,169	7,692	63%	62%	63%	82%	34%	33%	33%	54%

405 resolved conflicts to test RPREDICTOR. We intentionally used older data for training and newer data for testing. This is because such a setting can mimic the real-world scenarios, where RPREDICTOR can only refer to a project’s history data to suggest resolutions for future conflicts of that project.

5.2.1. Baseline

410 No prior work predicts developers’ resolution preferences, so we could not compare RPREDICTOR with any existing tool. However, we were still interested in how RPREDICTOR compares with a weighted random predictor. Thus, we created a **baseline** technique. We assumed that baseline somehow knows the ratios of conflicts separately resolved via KL, KR, or ME, and randomly predicts a label each time based on those ratios. As shown in Table 5, in the test set, there are 2,936, 1,979, and 2,777 conflicts separately resolved via KL, KR, and ME. Therefore, given a conflict, baseline predicts KL with a 38% probability (*i.e.*, $2936/(2936+1979+2777)$), and predicts KR and ME with 26% and 36% probabilities, respectively. Notice that the baseline technique is stronger than a naïve random classifier that predicts all resolutions with equal possibilities 420 (*i.e.*, 33%). In reality, it is also hard for any classifier to foresee the conflict distribution among all strategies. We made such a strong assumption to ensure that baseline is nontrivial, and to check whether RPREDICTOR outperforms it.

5.2.2. Comparison with Baseline

425 Table 5 counts the predictions of both RPREDICTOR and baseline for individual resolution strategies. According to the table, RPREDICTOR correctly labeled 1,815, 931, and 2,060 conflicts with KL, KR, ME, respectively. Meanwhile, baseline correctly labeled only 1,002, 649, and 901 conflicts with KL, KR, ME, respectively. These observations mean that RPREDICTOR predicts resolutions with much higher accuracies than baseline. 430

With the numbers reported in Table 5, we further measured effectiveness for both techniques using the metrics described in Section 5.1. As shown in Table 6, RPREDICTOR outperformed baseline for all metrics. For instance, for conflicts resolved by KR, RPREDICTOR achieved 58% precision, 47% recall, and 52% F-score; meanwhile, baseline only obtained 25% precision, 32% recall, and 28% 435 F-score. RPREDICTOR showed an overall effectiveness of 63% precision, 62%

Table 7: Effectiveness measurements for cross-project prediction

Experiment Id (Testing Fold #)	RPredictor				Baseline			
	P	R	F	C	P	R	F	C
1	46%	46%	46%	77%	34%	34%	34%	59%
2	49%	51%	50%	81%	35%	35%	35%	56%
3	47%	47%	47%	79%	33%	34%	33%	58%
4	50%	50%	50%	75%	34%	35%	34%	57%
5	41%	41%	41%	64%	38%	35%	36%	60%
6	42%	44%	43%	75%	34%	33%	34%	55%
7	44%	47%	46%	78%	36%	34%	35%	53%
8	47%	49%	48%	79%	36%	35%	36%	55%
9	50%	50%	50%	77%	34%	34%	34%	57%
10	44%	48%	46%	76%	34%	32%	33%	52%
Overall (All folds)	46%	47%	46%	76%	34%	34%	34%	57%

recall, 63% F-score, and 82% C-score; in contrast, baseline provided an overall effectiveness of 34% precision, 33% recall, 33% F-score, and 54% C-score. Both techniques worked more effectively to predict KL and ME, than to predict KR. This may be because there are fewer conflicts in the training set that were actually resolved by KR.

Finding 3: For within-project prediction, RPREDICTOR’s overall effectiveness measurements include 63% precision, 62% recall, 63% F-score, and 82% C-score. It outperformed baseline.

5.3. RQ2: Effectiveness of Cross-Project Prediction

In this experiment, we evaluated the real-world scenarios where a given project has little version history for RPREDICTOR to leverage. In such scenarios, RPREDICTOR can train a classifier with the conflict data from other repositories, and use that classifier to predict resolutions for the given project. We conducted 10-fold cross validation to evaluate RPREDICTOR’s effectiveness. Namely, we divided the 482 software projects randomly into 10 groups roughly evenly. For each group $G_i (i \in [1, 10])$, we ran an experiment by using the conflict data in the remaining nine groups for training, and adopting the data in G_i for testing. We calculated the effectiveness measurements for each of the 10 runs, and then computed the aggregated metrics of P, R, F, C among all runs.

5.3.1. Baseline

Similar to what we did for RQ1 (Section 5.2.1), we also created a weighted random classifier for cross-project prediction. In each of the 10 experiments mentioned above, baseline did not involve any training. Instead, it randomly assigned labels to conflicts based on the conflict distribution among three strategies in the test set. By empirically comparing RPREDICTOR with baseline, we explored how RPREDICTOR improves over weighted random prediction.

5.3.2. Comparison with Baseline

As shown in Table 7, RPREDICTOR outperformed baseline for all metrics in all 10 experiments. By aggregating our measurements for all folds, we got the overall effectiveness of RPREDICTOR as 46% precision, 47% recall, 46% F-score, and 76% C-score. Meanwhile, the overall effectiveness of baseline is 34%

precision, 34% recall, 34% F-score, and 57% C-score. Due to the space limit, we do not present tools’ effectiveness measurements for each resolution strategy. However, when we checked the detailed results for each strategy, we noticed that both tools predicted ME more accurately than predicting the other two strategies. In particular, RPREDICTOR always predicted ME more accurately than baseline; in 8 out of 10 experiments, RPREDICTOR suggested KL more accurately than baseline; in 9 out of 10 experiments, baseline suggested KR more accurately than RPREDICTOR.

Finding 4: *In cross-project prediction, RPREDICTOR achieved 41%–50% precision, 41%–51% recall, 41–50% F-score, and 64%–81% C-score. It outperformed baseline for all studied folds.*

5.3.3. Comparison between Cross-Project and Within-Project Prediction

We also compared RPREDICTOR’s cross-project prediction results (see Table 7) against its within-project prediction results (see Table 6). Generally speaking, both experiments have very similar data-splitting methodologies: they both use 90% of data (*i.e.*, conflicts or projects) for training and use 10% of data for testing. Nevertheless, RPREDICTOR predicted resolutions more effectively in the within-project setting, for all metrics. This may be because it is easier to predict the future resolution strategies of developers based on their resolution decisions for old conflicts. In contrast, it may be relatively harder to predict these developers’ resolution strategies based on the resolution decisions made by other developers in other projects. We also noticed that baseline achieved very similar effectiveness for the within-project and the cross-project settings. This is because the baseline technique does not have a training step. Its predictions are purely based on the random guesses derived from distributions of resolution strategies in test sets. No matter what data distribution we have for any test set, the random guesses typically achieve 33-34% overall F-scores.

Finding 5: *RPREDICTOR predicted resolutions more effectively in the within-project setting than in the cross-project setting.*

5.4. RQ3: Prediction Effectiveness on Unbalanced Data

As mentioned in Section 2, we used the conflict data of 100 repositories to characterize conflicts, and adopted the conflict data of another 482 repositories to train and test RPREDICTOR. All these 582 repositories have balanced distributions of different resolution strategies, which imply that developers did not show strong personal biases towards certain strategies; instead, they might decide upon resolutions solely based on branch edits, program context, and software evolution. To further investigate how effectively RPREDICTOR works given unbalanced data, we conducted another experiment. Specifically, among the 2,122 (*i.e.*, 2,731-609) repositories discarded in Section 2 due to the unbalanced distribution of different resolution strategies, we picked the most popular 100 repositories based on their star counts on GitHub, and experimented with them for both within-project and cross-project prediction. To facilitate discussion, Figures 5 and 6 separately visualize the overall distributions of resolution

strategies in the 582 balanced repositories and 100 unbalanced ones. As shown in Figure 6, the unbalanced data has the majority of conflicts (61%) resolved via KL, and least conflicts (15%) resolved via ME. Meanwhile, the balanced data has 35%, 27%, and 38% of conflicts separately resolved via KL, KR, and ME.

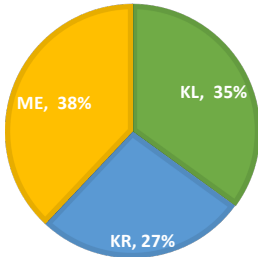


Figure 5: The resolution distributions among 582 balanced repositories

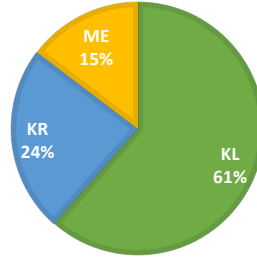


Figure 6: The resolution distributions among 100 unbalanced repositories

Table 8: RPREDICTOR’s effectiveness of within-project prediction given unbalanced data

	P	R	F	C
KL	86%	91%	89%	-
KR	79%	76%	77%	-
ME	64%	53%	58%	-
Overall	81%	69%	74%	86%

5.4.1. Effectiveness of Within-Project Prediction on Unbalanced Data

Similar to what we did for Section 5.2, in each of the 100 repositories with unbalanced data, we used the oldest 90% of resolved conflicts to train RPREDICTOR and used the remaining conflicts for testing. Table 8 shows our experiment results. By comparing this table against Table 6, we observed that RPREDICTOR worked much better when given unbalanced data for within-project prediction. Among the 100 repositories, it achieved 81% precision, 69% recall, 74% F-score, and 86% C-score; all the measurements are higher than those calculated for the balanced dataset (*i.e.*, 63%, 62%, 63%, 82%). In particular, RPREDICTOR obtained as high as 91% recall when predicting KL in the unbalanced dataset, probably because developers demonstrate very strong biases towards KL in that dataset and thus make that strategy easier to predict.

Finding 6: For within-project prediction tasks, RPREDICTOR predicted resolutions more effectively in the unbalanced dataset than in the balanced dataset.

5.4.2. Effectiveness of Cross-Project Prediction on Unbalanced Data

As with what we did for Section 5.3, we randomly split the 100 repositories into 10 groups with each group having 10 repositories, and performed 10-fold cross validation. As shown in Table 9, overall, RPREDICTOR achieved 53% precision, 43% recall, 47% F-score, and 49% C-score. Meanwhile, its overall metrics in the balanced dataset include 46% precision, 47% recall, 46% F-score, and 76% C-score (see Table 7). Given unbalanced data, RPREDICTOR obtained roughly the same F-score but a much lower C-score than what it did given balanced data; unbalanced data makes cross-project resolution prediction even

harder. Namely, if developers show extreme personal biases towards distinct resolution strategies in different projects, it can be very challenging to correctly predict the resolution strategies in one project based on strategies observed in other projects. Actually, among the 10 groups of our unbalanced dataset, there are 4 groups with strong preferences towards KL (*i.e.*, over 50% of conflicts were resolved via KL) and 3 groups with strong biases towards KR. The classifiers trained with such unbalanced data predict KL or KR most of the times but seldom predict ME, although ME is a more conservative strategy than KL and KR. Consequently, such classifiers earn much lower conservativeness scores.

Table 9: RPREDICTOR’s effectiveness of cross-project prediction given unbalanced data

Experiment Id (Testing Fold #)	P	R	F	C
1	41%	42%	41%	55%
2	63%	44%	52%	47%
3	42%	40%	41%	48%
4	52%	46%	49%	51%
5	44%	37%	41%	52%
6	52%	46%	49%	57%
7	34%	31%	33%	50%
8	38%	30%	33%	38%
9	47%	42%	44%	53%
10	51%	38%	43%	59%
Overall (All folds)	53%	43%	47%	49%

Finding 7: For cross-project prediction tasks, RPREDICTOR predicted resolutions less conservatively in the unbalanced dataset than in the balanced one.

5.5. RQ4: Sensitivity to The Amount of Training Data

In our experiment settings, by default, we typically used 90% of overall data for training and 10% of data for testing. However, it is unknown how the amount of training data can influence RPREDICTOR’s effectiveness. Therefore, we performed another experiment of within-project prediction, by tuning the amount of training data in use. Specifically, in the balanced dataset (*i.e.*, 482 repositories), we split the conflict data of each repository into 11 portions evenly (each portion having the same number of conflicting chunks): p_1, p_2, \dots, p_{11} . Here, p_1 represents the oldest data portion in history and p_{11} is the newest one. We trained and tested RPREDICTOR 10 times, with each of the iterations using p_{11} as the testing data but using a distinct set of portions for training. As shown

Table 10: RPREDICTOR’s effectiveness of within-project prediction, when different amounts of training data are provided in different iterations

Iteration Id	Data Portions		RPredictor			
	Training	Testing	P	R	F	C
1	p_{10}	p_{11}	42%	45%	43%	66%
2	p_9, p_{10}	p_{11}	51%	52%	51%	71%
3	p_8-p_{10}	p_{11}	54%	55%	54%	72%
4	p_7-p_{10}	p_{11}	52%	52%	52%	72%
5	p_6-p_{10}	p_{11}	54%	54%	54%	75%
6	p_5-p_{10}	p_{11}	55%	56%	56%	75%
7	p_4-p_{10}	p_{11}	56%	56%	56%	76%
8	p_3-p_{10}	p_{11}	56%	56%	56%	75%
9	p_2-p_{10}	p_{11}	56%	56%	56%	75%
10	p_1-p_{10}	p_{11}	56%	56%	56%	76%

in Table 10, the 1st iteration adopts p_{10} for training; the 2nd iteration exploits
 555 both p_9 and p_{10} to train RPREDICTOR; the 10th iteration uses 10 portions p_1 – p_{10}
 in training.

RPREDICTOR’s effectiveness increases or roughly remains the same when
 the amount of training data grows. Specifically when only p_{10} was provided,
 RPREDICTOR obtained 42% precision, 45% recall, 43% F-score, and 66% C-
 560 score. During the first three iterations, as the training data increased from
 one portion to three portions, all measurements increased steadily. Meanwhile,
 during the last six iterations, while the training data increased from five to ten
 portions, RPREDICTOR’s effectiveness stabilized without much change. One
 possible reason to explain the observed increase is that when training data is
 565 insufficient, providing more data enables RPREDICTOR to better characterize
 diverse conflicting scenarios and thus better predict resolutions. However, once
 the training data is sufficient, offering more data does not necessarily improve
 RPREDICTOR’s effectiveness. Consequently, all measurements stabilize. Based
 on this experiment, we decided for our other experiments (except for RQ4 and
 570 RQ5), by default, we used 90% of data for training and 10% of data for testing,
 in order to train RPREDICTOR with sufficient data and to observe the best
 effectiveness measurements achievable by RPREDICTOR.

Finding 8: *RPREDICTOR’s effectiveness improves or stabilizes when more training data is provided.*

5.6. RQ5: Sensitivity to The Age of Training Data

575 When looking at Table 10, one may be tempted to wonder whether the age
 of training data also influences RPREDICTOR’s effectiveness. Actually, between
 different iterations shown in Table 10, both the (1) age and (2) amount of train-
 ing data are different. To explore the influence of each factor, we conducted
 an additional experiment with the 11 data portions mentioned in Section 5.5
 580 (each portion having the same number of conflicting chunks). In this exper-
 iment, we repetitively trained RPREDICTOR with a distinct data portion but
 always tested it with p_{11} . As shown in Table 11, the 1st iteration uses p_{10} —
 the youngest portion within $[p_1, p_{10}]$ —as the training data; the 2nd iteration
 uses p_9 ; the 10th iteration uses the oldest data p_{10} . Because the training data
 585 in each iteration has roughly equal numbers of data points, the comparison of
 effectiveness measurements across iterations reflects the impact of data age.

Table 11: RPREDICTOR’s effectiveness of within-project prediction, when differently aged data is provided for training

Iteration Id	Data Portions		RPredictor			
	Training	Testing	P	R	F	C
1	p_{10}	p_{11}	43%	44%	43%	66%
2	p_9	p_{11}	48%	49%	49%	69%
3	p_8	p_{11}	45%	48%	46%	68%
4	p_7	p_{11}	42%	43%	43%	74%
5	p_6	p_{11}	45%	47%	46%	81%
6	p_5	p_{11}	34%	38%	36%	70%
7	p_4	p_{11}	37%	41%	39%	69%
8	p_3	p_{11}	44%	43%	44%	80%
9	p_2	p_{11}	42%	44%	43%	83%
10	p_1	p_{11}	39%	42%	41%	63%

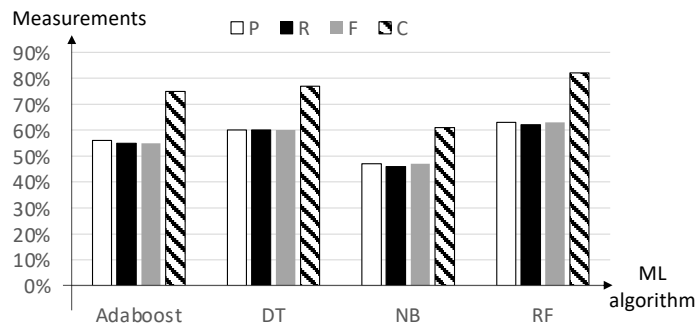


Figure 7: RPRECICTOR using different ML algorithms for within-project prediction

According to Table 11, as the training data gets older, the effectiveness measurements either increase or decrease, without presenting a consistent change trend. For instance, in the 1st iteration, RPRECICTOR obtained 43% precision, 44% recall, 43% F-score, and 66% C-score. In the 9th iteration, RPRECICTOR achieved a slightly lower precision (42%), the same recall (44%), the same F-score (43%), but the highest C-score (83%). However, in the 10th iteration, it acquired the lowest measurements: 39% precision, 42% recall, 41% F-score, and 63% C-score. The phenomena imply that data age does not have a consistently positive or negative impact on prediction results. The prediction effectiveness increased probably because the training data became more similar to the testing data, and decreased probably due to the less similarity between training and test data. Therefore, both the consistent effectiveness improvements and stabilized measurements we observed in Table 10 are mainly contributed by the increase of training data, instead of data aging.

Finding 9: *RPRECICTOR's effectiveness does not consistently change with the age of training data.*

5.7. RQ6: Sensitivity to The Adopted Machine-Learning Algorithm

When designing RPRECICTOR, we did not know what machine-learning (ML) algorithm was more suitable. Thus, we experimented with four ML algorithms in both the within-project and cross-project settings, to observe how RPRECICTOR's effectiveness varies with the adopted algorithm. We studied Adaboost, decision tree (DT), naïve bayes (NB), and random forest (RF). As mentioned in Section 5.2, for the within-project setting, we used 90% of the oldest resolved conflicts in each project's version history for training, and 10% of conflicts (the most recent ones) for testing. For the cross-project setting, we used 10-fold cross validation (as in Section 5.3). In all of our experiments, we leveraged the ML implementation provided by scikit-learn [55], and used the default parameter settings for all adopted ML algorithms.

As shown in Figures 7 and 8, RPRECICTOR achieved the highest effectiveness when using RF. For within-project prediction, RF obtained 63% precision, 62% recall, 63% F-score, and 82% C-score. DT had lower effectiveness than RF, but better than the other two alternatives; it obtained 60% precision, 60%

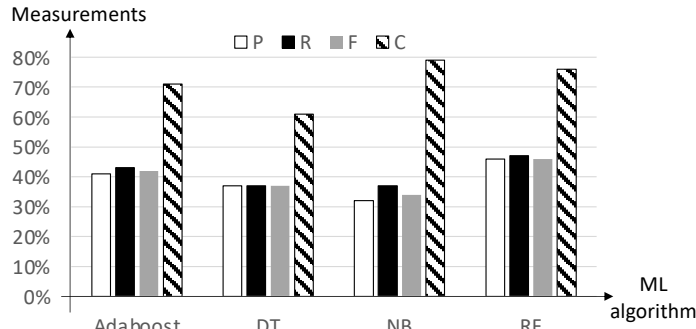


Figure 8: RPRELECTOR using different ML algorithms for cross-project prediction

recall, 60% F-score, and 77% C-score. NB was the least effective and got 47%
 620 precision, 46% recall, 47% F-score, and 61% C-score. For cross-project prediction, RF obtained 46% precision, 47% recall, 46% F-score, and 76% C-score. Adaboost performed worse than RF; it got 41% precision, 43% recall, 42% F-score, and 71% C-score. NB achieved the most interesting results. Among the four algorithms studied, NB acquired the lowest precision (32%), lowest recall
 625 (37%), and lowest F-score (34%); nevertheless, it acquired the highest C-score (79%). This is mainly because NB predicted a lot more ME resolutions than the other algorithms. Comparing the effectiveness of distinct algorithms in both within-project and cross-project settings, we decided to use RF as the default ML algorithm in RPRELECTOR because RF often outperformed the others.

630 **Finding 10:** *Among the four experimented machine learning algorithms, RF generally outperformed the others when being used in RPRELECTOR.*

5.8. RQ7: Sensitivity to Threshold Setting

In the experiments mentioned above, the highest C-score RPRELECTOR achieved is 82%. It means that 82% of the resolution strategies recommended by RPRELECTOR are conservative; in other words, they correctly predict the
 635 developers’ preference, or ask developers to resolve the conflict manually. However, some developers may prefer RPRELECTOR to provide lower C-scores (*i.e.*, to predict more KL or KR labels) in order to save more effort, even if the predictions are more risky or less precise. Such preferences are meaningful for projects with very good test suites, in which developers can trust automated testing to reliably
 640 decide the correctness of any program version whose conflicts were automatically resolved. Other developers may prefer RPRELECTOR to achieve higher C-scores (*i.e.*, to predict ME more often) in order to avoid prediction errors, even though the predictions save less effort. Such preferences are important for projects with very limited test suites, in which developers cannot blindly trust
 645 automated testing to always validate the correctness of programs.

To give developers more control over RPRELECTOR’s predictions, we created a configurable variant of RPRELECTOR—RPRELECTOR_v, which offers a parameter th_M so that developers can fine-tune automatic prediction based on their relative tolerance for incorrect KL or KR predictions.

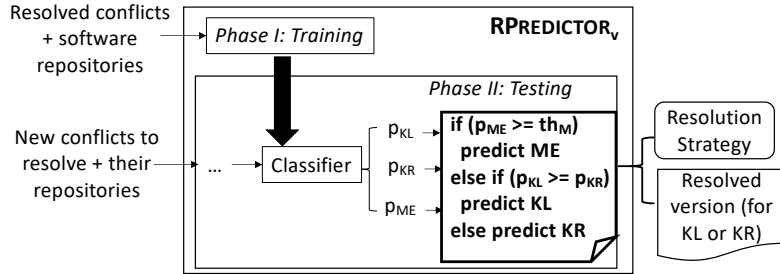


Figure 9: RPRELECTOR_v—our customizable variant of RPRELECTOR, which uses a threshold th_M to fine-tune the prediction results

5.8.1. A Threshold-Based Variant Approach: RPRELECTOR_v

Figure 9 shows our approach for RPRELECTOR’s customizable variant. Similar to RPRELECTOR, this variant also trains a classifier to predict the resolution strategy for any given merge conflict. However, this variant now allows its users to increase (or decrease) its prediction preference for ME. Given a merge conflict and its related software repository, a classifier generates three predicted likelihoods: p_{KL} , p_{KR} , and p_{ME} . These likelihoods indicate how likely the predictor believes that the conflict should be resolved via KL, KR, or ME. All likelihoods vary within $[0, 1]$; $p_{KL} + p_{KR} + p_{ME} = 1$. The original approach RPRELECTOR returns its prediction based on the highest likelihood among p_{KL} , p_{KR} , and p_{ME} . In contrast, the customizable variant RPRELECTOR_v first compares p_{ME} with the user-configured threshold th_M . As shown in Figure 9, if $p_{ME} \geq th_M$, then RPRELECTOR_v predicts ME; otherwise, it predicts one of the other two strategies, the one with the higher likelihood (KL or KR).

In this way, developers can modify th_M to tune RPRELECTOR_v’s conservativeness. When $th_M = 0$, it predicts all conflicts conservatively as ME. In this scenario, developers would not get any incorrect KL or KR predictions, but they would not benefit from RPRELECTOR_v automatically acting on the KL or KR predictions (*i.e.*, it would not save effort). On the other extreme, when $th_M = 1.0$, all conflicts are predicted to resolve via either KL or KR. In this scenario, RPRELECTOR_v would save developers high effort (it would automatically resolve all conflicts by KL or KR), but some of those KL or KR resolutions would not be what the developers preferred (they would be incorrect predictions). With other values of th_M , developers can decide their own personal middle-ground between these two extreme points.

5.8.2. Experiment with RPRELECTOR_v

To study the trade-offs between F-score, C-score, and the potential effort-saving by automatic resolution that developers could obtain with RPRELECTOR_v, in this experiment we tuned th_M from 0.1 to 1, with 0.1 increments. For each threshold setting, we applied RPRELECTOR_v to perform both within-project and cross-project prediction. For this section, we defined another metric to measure the potential effort-saving by automatic resolution:

Effort-saving (E) Score or E-score measures among all predictions, for how many of them RPRELECTOR_v outputs KL or KR and automatically resolves

the conflict. The score is within [0%, 100%].

$$E = \frac{\# \text{ of conflicts automatically resolved via KL or KR}}{\text{All predictions}} \quad (7)$$

685 Figure 10 shows RPREDICTOR_v's performance for within-project prediction. As th_M increased, C-score consistently decreased and E-score increased. F-score was stable when $th_M \in (0, 0.7]$; it decreased as th_M increased from 0.7 to 1. For the most conservative threshold ($th_M = 0.1$), RPREDICTOR_v labeled many conflicts with ME; it only labeled them KL or KR when the predicted likelihoods
690 were very high (RPREDICTOR_v was quite sure about those predictions). In this scenario, RPREDICTOR_v achieved a C-score of 94%, E-score of 34%, and F-score of 68%. This shows that RPREDICTOR_v can achieve as much as 34% effort savings (E-score) by also very rarely predicting KL or KR incorrectly (with very high C-score). For the most liberal threshold ($th_M = 1.0$), RPREDICTOR_v
695 labeled no conflict with ME. Instead, it only produced KL and KR labels to automate all resolutions. In such scenarios, RPREDICTOR_v incorrectly labeled many conflicts as KL or KR, applying a strategy that was not preferred by the developers. Consequently, the achieved C-score was 50%, E-score was 100%, and F-score was 38%. This option would save all the effort of conflict resolution, but
700 it would likely require additional mechanisms to detect incorrectly applied KL or KR resolutions, *e.g.*, using a very strong test suite that is either manually crafted or automatically generated (*e.g.*, via good fuzzy testing techniques).

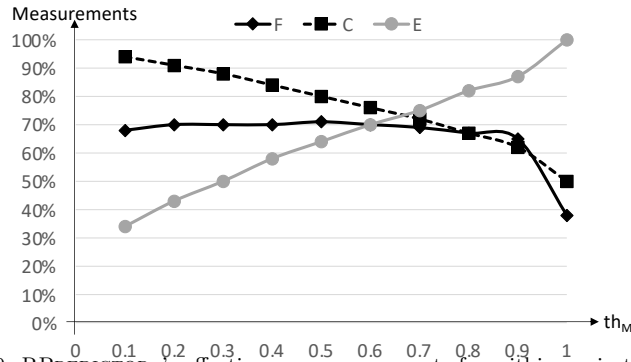


Figure 10: RPREDICTOR_v's effectiveness measurements for within-project prediction

We believe that other intermediate thresholds would be more popular. Between $th_M = 0.1$ and $th_M = 1.0$, F-score was stable initially and then decreased.
705 As th_M increased, RPREDICTOR_v achieved different trade-offs between precision and recall for each strategy. Figure 10 also shows that developers could achieve increasing effort savings (E-score), at the cost of accepting increasing ratios of incorrect KL or KR predictions (lower C-scores). However, it is also worth noting that E-scores grew faster than C-scores fell, which means that multiple
710 intermediate thresholds may be attractive for different developers. For example, the thresholds in $(0, 0.5]$ achieved up to 64% effort savings with C-scores no lower than 80%.

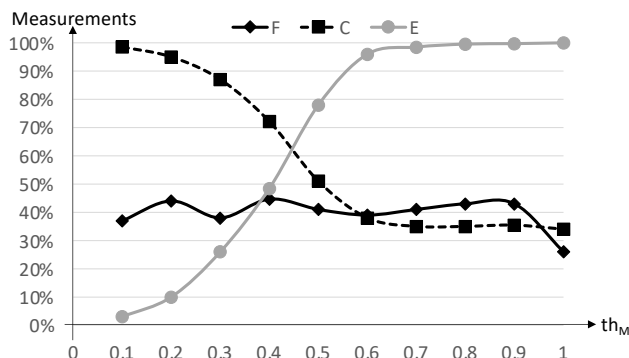


Figure 11: RPRELECTOR_v's effectiveness measurements for cross-project prediction

Figure 11 shows RPRELECTOR_v's performance for cross-project prediction. As th_M increased, C-score decreased first and then stabilized when $0.7 \leq th_M \leq 1$; E-score increased first and then stabilized when $0.7 \leq th_M \leq 1$. F-score vibrated in the range [37%, 45%] when $th_M \leq 0.9$, and dropped afterwards. We saw in RQ2 that RPRELECTOR's predictions are less effective in the cross-project setting than in the within-project setting. This is also reflected by Figures 10 and 11, since RPRELECTOR_v generally produced a worse trade-off between effort savings and conservativeness. In cross-project prediction, the most conservative threshold ($th_M = 0.1$) provided almost no effort saving, and if we wanted to keep C-score over 80%, we could only achieve up to 26% effort savings ($th_M \leq 0.3$).

Finding 11: RPRELECTOR_v generally achieved better trade-offs between effort savings and conservativeness in the within-project setting than in the cross-project setting. For within-project prediction, RPRELECTOR_v could save up to 63% of efforts by lowering the C-score while keeping it above 80%.

6. Threats to Validity

Threats to External Validity. Our characterization study investigates 12 candidate features, which are defined either based on prior studies or our insights. It is possible that there are other features (*e.g.*, types of edits in branches) that are potentially correlated with developers' resolution strategies, and can be leveraged to better predict resolutions. In the future, we plan to define and explore more candidate features, so that our characterization study is more representative. By revealing and incorporating new features, we can also strengthen the prediction capability of RPRELECTOR. Our study and experiments are done on Java projects, although the methodology is generally applicable to programs written in any language. It is possible that the results of our study and evaluation do not generalize well to programs written in other languages. In the future, we plan to conduct larger-scale experiments to include non-Java programs.

Threats to Construct Validity. When crawling the owner developers of commits in software repositories, we assumed that there is one-to-one mapping relationship between developers and user IDs (*i.e.*, email addresses). Namely, we

assumed that each developer has only one user ID, which is not shared with any other developer. However, in reality, it is possible that a developer leverages multiple user IDs when checking in different commits (*i.e.*, one-to-many), while some developers share a single user ID when committing program changes (*i.e.*, many-to-one). Such one-to-many and many-to-one relations between developers and user IDs can make our data analysis imprecise. However, we believe that the corner cases of one-to-many and many-to-one mappings are rare, causing little impact on our research findings.

7. Discussion

In this section, we discuss various aspects of our approach to further clarify its applicability.

7.1. The Benefit of RPREDICTOR's Recommendations for Developers

Given a merge conflict, RPREDICTOR predicts the resolution strategy, and even recommends a merged version if the predicted strategy is KL or KR. Readers may be tempted to underestimate the usefulness of RPREDICTOR, because KL and KR seem much simpler to execute than ME. However, we argue that conflict resolution involves not only resolution implementation, but also decision-making; RPREDICTOR helps considerably reduce the manual effort on the decision-making process.

Prior work [49] mentions that 56% of developers have deferred at least once when responding to a merge conflict, which makes conflict resolution more complex as time passes; the key challenges that developers have to overcome when trying to resolve conflicts include (1) understanding the conflicting code, and (2) getting enough metadata information about the conflict (e.g., who made the change, why, and when). RPREDICTOR characterizes software conflicts from 12 distinct aspects, in order to automatically comprehend conflicts and retrieve metadata information related to those conflicts. Therefore, when RPREDICTOR correctly predicts KL and KR, developers do not need to go through the painful process of conflict comprehension and resolution.

As multiple studies [64, 19, 53] show that the majority of conflicts are resolved via KL and KR, RPREDICTOR's good precision of predicting KL/KR can significantly save developers effort, effort that would have otherwise been spent to manually analyze and resolve such conflicts.

7.2. The Impact of Mispredictions and Developers' Trust in Automated Recommendations

If RPREDICTOR predicts ME and requires developers to manually resolve some conflicts, developers cannot save any manual effort, but it also does not put any extra effort on those conflicts, either.

In the scenarios when RPREDICTOR incorrectly predicts KL or KR, developers may need to put extra effort to examine the tool-suggested strategies. However, if the test cases in software projects (1) have sufficient coverage, (2)

do not conflict with each other across branches, and (3) reliably express the intended behaviors of merged software, developers do not need to spend more time reasoning about whether tool-generated resolutions work. Instead, they
785 can rely on testing to validate automated resolutions.

Furthermore, developers may choose to manually double-check if they personally agree with RPREDICTOR’s prediction before applying it, which can reduce its ratio of mispredictions. This mode of operation would imply a lower effort reduction for developers, but it can still be more efficient than reviewing
790 all the details of the merge conflict.

At the end of the day, we expect different developers to show different preferences in terms of how liberally they want to directly apply RPREDICTOR’s recommendations. That is why we proposed a variant of RPREDICTOR in Section 5.8, that gives them flexibility to make RPREDICTOR provide predictions
795 that save higher effort producing more mispredictions, the opposite trade-off, or other points in between.

In future work, we will also explore how to use *explainable machine learning approaches* to increase the trust of RPREDICTOR’s recommendations for developers, trying different approaches for explaining why RPREDICTOR is recommending a particular strategy.
800

7.3. Applicability of RPREDICTOR on Less-Balanced Projects

We evaluated RPREDICTOR in a dataset of projects that resolved merge conflicts in a relatively balanced way, *i.e.*, all decisions were taken with relatively similar frequencies. We did this intentionally to evaluate RPREDICTOR in the
805 kinds of projects for which we estimate they would benefit from it most: those projects which do not have a very clear *typical* way to resolve conflicts, *i.e.*, those in which no choice is strongly overrepresented.

However, we believe that RPREDICTOR could also benefit projects in which KL or KR is the *typical* choice to resolve merge conflicts, *i.e.*, in which that
810 strategy is chosen the majority of the time. In such cases, developers would also benefit from RPREDICTOR, because it will capture this bias in its training and it will in fact predict resolution strategies with higher accuracy. We performed an experiment showing RPREDICTOR’s higher accuracy in an unbalanced dataset in Section 5.4 (RQ3).

The only case of projects that would not benefit as much from RPREDICTOR are those which choose ME to resolve their merge conflicts the majority of the time — since RPREDICTOR’s recommendations are most beneficial when it predicts KL or KR. However, such situations are less common — past work
[69, 27] showed that KL and KR are the most popular strategies to resolve
820 merge conflicts.

7.4. What If A Project Has Little Training Data Available?

When using RPREDICTOR, users do not have to train RPREDICTOR on a large dataset of software repositories. Instead, for the within-project setting, they can use all conflicts extracted from one project’s version history for classifier training, and leverage that trained classifier to predict resolutions for any
825

new conflicts in the same project. For the cross-project setting, users can simply use the trained classifier open-sourced on our project website [3], instead of training any classifier from scratch. In order to help users decide whether RPREdictor should perform within- or cross- project prediction for their circumstances, we actually ranked the 482 experimented repositories in descending order of the number of conflicting chunks they contain in version history. From that ranked list, we sampled the 1st project (the one with the most conflicts), the 482th project (the one with fewest conflicts), and 9 projects standing between at roughly 10%-interval of ranks. Table 12 shows all the sampled 11 projects, the total number of conflicts contained by each project, the number of conflicts used for training (*i.e.*, 90% of the total), and the F-scores achieved by RPREdictor for within-project prediction.

Table 12: RPREdictor’s F-scores for 11 sampled projects for within-project prediction

Rank	Total # of conflicts	# of conflicts used for training	F-score
1 st	5,114	4,603	92%
48 th	405	365	58%
96 th	150	135	45%
144 th	85	77	37%
192 th	62	56	27%
240 th	44	40	27%
288 th	30	27	17%
336 th	24	22	22%
384 th	17	15	0%
432 th	11	10	0%
482 th	11	10	0%

According to this table, as training data decreases, F-score generally decreases or stabilizes; this trend coincides with our observation in Section 5.5. The phenomenon implies that if a user’s software repository has a few resolved conflicts (*e.g.*, less than 135), she/he can consider using cross-project prediction as the conflicts in version history seem insufficient to train a good within-project predictor. Otherwise, if the user’s software repository has sufficient resolved conflicts (*e.g.*, hundreds or even thousands of conflicts), she/he can apply RPREdictor to do within-project prediction for better accuracy.

8. Related Work

Our research is related to empirical studies on merge conflicts, awareness-raising tools, and automated software merge.

8.1. Empirical Studies on Merge Conflicts

Several studies were conducted to characterize the relationship between merge conflicts and other aspects of software maintenance [25, 9, 42, 45, 52]. For instance, Estler *et al.* [25] surveyed 105 student developers, and found that the lack of awareness (*i.e.*, knowing “who’s changing what”) occurs more frequently than merge conflicts. Leßenich *et al.* [42] surveyed 41 developers and identified 7 potential indicators (*e.g.*, number of changed files in both branches)

for merge conflicts. With further investigation of the indicators, the researchers found that none can predict the conflict frequency. Similarly, Owhadi-Kareshk *et al.* defined nine features (*e.g.*, number of added and deleted lines in a branch) to characterize merging scenarios; they trained a machine-learning model that
860 predicts conflicts with 57%–68% accuracy [52].

Similar to these studies, our study also characterizes merge conflicts. However, it is different in two aspects. First, our study explores how different features characterize developers’ strategies of conflict resolution. Second, our study motivates our research to automatically predict resolution strategies, while
865 existing studies motivate research to automatically predict conflict occurrence.

Some other studies characterize the root causes and/or resolutions of *textual* conflicts [69, 50, 27, 15, 53]. Specifically, Yuzuki *et al.* inspected hundreds of textual conflicts [69]. They observed that conflicting updates caused 44% of conflicts to the same line of code, and developers resolved 99% of conflicts by
870 taking either the left- or right- version of code. Brindescu *et al.* [15] manually inspected 606 textual conflicts. They characterized merge conflicts in terms of the AST diff size, LOC diff size, and the number of authors. They identified three resolution strategies: SELECT ONE (*i.e.*, keep edits from one branch), INTERLEAVE (*i.e.*, keep edits from both sides), and ADAPTED (*i.e.*, change
875 existing edits and/or add new edits). Pan *et al.* [53] explored the merge conflicts in Microsoft Edge; they classified those conflicts based on file types, conflict locations, conflict sizes, and conflict-resolution patterns. Driven by their empirical study, the researchers further investigated to use program synthesis for conflict resolution. The prototype of their resolution tool only tries to concatenate edits
880 from both branch versions, incapable of suggesting KL or KR resolutions.

These studies inspired us to define and study candidate features that may help predict developers’ resolution strategies for conflicts. However, none of these studies conduct statistical analysis between any recognized features and developers’ resolutions; our study performed that analysis.

885 8.2. Awareness-Raising Tools

Tools [56, 63, 14, 17, 28, 18, 40, 37, 44] were created to monitor and compare programmers’ development activities, in order to improve team activity awareness. For instance, CASI [63] and Palantír [56] inform a developer of the artifacts changed by other developers, calculate the severity of those changes,
890 and visualize the information. Cassandra [37] is a conflict minimization technique. It observes the super-sub and caller-callee dependencies between program entities. By treating those dependencies as constraints on file-editing tasks, Cassandra identifies tasks that will conflict when performed in parallel. It then appropriately schedules tasks to recommend conflict-free development paths.
895 Crystal [17, 18] and WeCode [28] proactively detect collaboration conflicts via speculative analysis. They eagerly merge the program changes applied to different software branches, even before those changes are all pushed to the master repository in the distributed version control system (DVCS). They leverage textual merge, automatic build, and automatic testing in sequence to reveal the
900 potential conflicts between branches.

The tools mentioned above can proactively detect and report merge conflicts. However, they do not characterize developers’ resolution preferences, neither do they automatically recommend any resolution strategy.

8.3. Automated Software Merge

905 Tools were proposed to detect or resolve merge conflicts [47, 13, 12, 41, 51, 19, 71, 66, 65, 2, 67, 70, 24]. Mens *et al.* [47] published a survey on software merging techniques. FSTMerge [13, 19, 2] parses code for ASTs, and matches nodes between L and R purely based on the class or method signatures; it then integrates the edits inside each pair of matched method nodes via textual merge. IntelliMerge [65] improves FSTMerge’s effectiveness by detecting and resolving
910 refactoring-related conflicts. Similar to FSTMerge, JDime [12, 41] also matches Java methods and classes based on syntax trees. However, JDime merges edits inside matched methods by matching and manipulating ASTs. AutoMerge [71] improves over JDime. When branch edits are incompatible with each other,
915 AutoMerge attempts to resolve conflicts by proposing alternative strategies to merge L and R, with each strategy integrating the edits between branches in distinct ways. SafeMerge [66] checks if a merging scenario introduced new semantics. RPREDICTOR complements all these techniques, as it models and predicts developers’ resolution preferences.

920 MergeHelper [51] records the chronological sequence of edit operations made by programmers on the Eclipse Java editor. Given two branch versions—L and R—that conflict with each other, MergeHelper explores the recorded edit sequences before both versions, to locate the most recent snapshot that appears in the evolution history and is consistent with L and R. In other words, Merge-
925 Helper rolls back edits applied by both branches, until finding an intermediate version that occurs just before the first conflict was introduced. It provides detailed edit information to help developers understand how conflicts got introduced, but does not suggest resolution strategies as RPREDICTOR does.

DeepMerge [24], MergeBERT [67], and GMerge [70] automatically resolve
930 conflicts using deep-learning methods. However, DeepMerge only focuses on conflicts with less than 30 lines [67]; it is not applicable to more complicated conflicts. Given a textual conflict, both DeepMerge and MergeBERT are designed to integrate partial edits from L and R for resolution, instead of proposing KL or KR. GMerge does not focus on textual conflicts; instead, it deals with
935 a different type of merge conflicts where conflicting edits can be co-applied to the merged version but trigger semantic errors. RPREDICTOR complements the learning-based approaches mentioned above. That is, RPREDICTOR can predict conflicts that get resolved by KL or KR (the majority, according to the literature), and when Rpredictor predicts ME, it can be complemented with an
940 alternative method (like DeepMerge or MergeBERT) to automate a resolution based on the combination of lines.

9. Conclusion

Software merge is complex and time-consuming. People defined the term “Integration Hell” to refer to the challenges of addressing merge conflicts. Although many tools were proposed to detect and even resolve merge conflicts, little tool support is available to automatically resolve conflicts by observing and mimicking developers’ resolution strategies. Consequently, existing tools mainly pinpoint issues of merge conflicts, rarely providing solutions to those issues. In this paper, we conducted the first characterization study to explore any statistical correlation between 12 features of merge conflicts and developers’ resolution strategies. Our study shows for the first time that all of the explored features can help predict developers’ resolution strategies.

Motivated by our study, we also designed and implemented a novel approach—RPREDICTOR—to predict developers’ resolution strategy, given a merge conflict and its related software repository. Our comprehensive evaluation of the tool with a large-scale dataset containing 74,861 resolved conflicts showed that RPREDICTOR effectively predicted resolutions. By training prediction models with the random forest (RF) algorithm, RPREDICTOR could achieve 63% precision, 62% recall, 63% F-score, and 82% C-score for within-project prediction; it also got 46% precision, 47% recall, 46% F-score, and 76% C-score for cross-project prediction. Our sensitivity analysis shows that compared with other machine-learning (ML) algorithms, RF achieved the best results when being used in RPREDICTOR; RPREDICTOR is sensitive to both the amount and age of training data; as more training data is provided, RPREDICTOR’s effectiveness increases or stabilizes. Developers can also customize RPREDICTOR’s th_M threshold to more or less often predict M resolutions, making it save less or more effort.

In the future, we will explore more features and more ML algorithms, to further improve the representativeness of our characterization study and to strengthen the capability of RPREDICTOR. For example, we will explore additional prediction features related to, *e.g.*, code-change history [58, 59, 57, 61, 62], testing activity, *e.g.*, [39, 26, 38], decision-making metadata, *e.g.*, [48, 11, 10], developer expertise, *e.g.*, [60, 20], build failure prediction *e.g.*, [32, 34, 33, 31, 35, 36], security issue prediction *e.g.*, [22, 30] or cross-language issues, *e.g.*, [23].

10. Research Artifact

We made available the research artifact for our paper [3].

Acknowledgement

We thank all reviewers for their valuable feedback. This work was partially funded by NSF CCF-1845446, NSF CCF-2046403, Universidad Rey Juan Carlos under the International Distinguished Researcher award C01INVEDIST, and by Saudi Arabian Cultural Mission (SACM). This work also served as foundation for award PID2022-142964OA-I00 by the Spanish Agencia Estatal de Investigación.

References

- 985 [1] 2021. git merge - Integrating changes from another branch. <https://www.git-tower.com/learn/git/commands/git-merge>.
- [2] 2021. jFSTMerge. <https://github.com/guilhermejccavalcanti/jFSTMerge>.
- [3] 2022. Research Artifact for Paper: Automatic Prediction of Developers' Resolutions for Software Merge Conflicts. <https://figshare.com/s/3b28e1917a7a05588891>.
990
- [4] 2023. aosp-mirror / platform_frameworks_base. https://github.com/aosp-mirror/platform_frameworks_base.
- [5] 2023. CCI-MIT/XCoLab. <https://github.com/CCI-MIT/XCoLab>.
- 995 [6] 2023. jgralab/jgralab. <https://github.com/jgralab/jgralab>.
- [7] 2023. jhy/jsoup. <https://github.com/jhy/jsoup>.
- [8] 2023. Kruskal-Wallis Test. <https://www.statisticssolutions.com/kruskal-wallis-test/>.
- [9] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma. 2017. An Empirical Examination of the Relationship between Code Smells and Merge Conflicts. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 58–67. <https://doi.org/10.1109/ESEM.2017.12>
1000
- [10] Khadijah Al Safwan, Mohammed Elarnaoty, and Francisco Servant. 2022. Developers' Need for the Rationale of Code Commits: An In-breadth and In-depth Study. *Journal of Systems and Software* (2022).
1005
- [11] Khadijah Al Safwan and Francisco Servant. 2019. Decomposing the Rationale of Code Commits: The Software Developer's Perspective. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
1010
- [12] Sven Apel, Olaf Lessenich, and Christian Lengauer. 2012. Structured Merge with Auto-tuning: Balancing Precision and Performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (Essen, Germany) (ASE 2012)*. ACM, New York, NY, USA, 120–129. <https://doi.org/10.1145/2351676.2351694>
1015
- [13] Sven Apel, Jorg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kastner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (Szeged, Hungary) (ESEC/FSE '11)*. ACM, New York, NY, USA, 190–200. <https://doi.org/10.1145/2025113.2025141>
1020

- 1025 [14] Jacob T. Biehl, Mary Czerwinski, Mary Czerwinski, Greg Smith, and George G. Robertson. 2007. FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) (*CHI '07*). ACM, New York, NY, USA, 1313–1322. <https://doi.org/10.1145/1240624.1240823>
- 1030 [15] Caius Brindescu, Iftekhhar Ahmed, Carlos Jensen, and Anita Sarma. 2020. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering* 25, 1 (2020), 562–590. <https://doi.org/10.1007/s10664-019-09735-4>
- 1035 [16] Caius Brindescu, Yenifer Ramirez, Anita Sarma, and Carlos Jensen. 2020. Lifting the Curtain on Merge Conflict Resolution: A Sensemaking Perspective. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 534–545. <https://doi.org/10.1109/ICSME46990.2020.00057>
- 1040 [17] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) (*ESEC/FSE '11*). ACM, New York, NY, USA, 168–178. <https://doi.org/10.1145/2025113.2025139>
- 1045 [18] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *IEEE Transactions on Software Engineering* 39, 10 (Oct 2013), 1358–1375. <https://doi.org/10.1109/TSE.2013.28>
- 1050 [19] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and Improving Semistructured Merge. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 59 (Oct. 2017), 27 pages. <https://doi.org/10.1145/3133883>
- [20] Lykes Claytor and Francisco Servant. 2018. Understanding and Leveraging Developer Inexpertise. In *International Conference on Software Engineering: Companion Proceedings*.
- 1055 [21] Catarina Costa, José J. C. Figueiredo, Gleiph Ghiotto, and Leonardo Gresta Paulino Murta. 2014. Characterizing the Problem of Developers' Assignment for Merging Branches. *International Journal of Software Engineering and Knowledge Engineering* 24 (2014), 1489–1508.
- 1060 [22] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

- 1065 [23] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-Use and Portability of Regular Expressions. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
- 1070 [24] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu Lahiri. 2023. DeepMerge: Learning to Merge Programs. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1599–1614. <https://doi.org/10.1109/TSE.2022.3183955>
- 1075 [25] H Christian Estler, Martin Nordio, Carlo A Furia, and Bertrand Meyer. 2014. Awareness and merge conflicts in distributed software development. In *2014 IEEE 9th International Conference on Global Software Engineering*. IEEE, 26–35.
- [26] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. 2017. An Empirical Study of Activity, Popularity, Size, Testing, and Stability in Continuous Integration. In *International Conference on Mining Software Repositories*.
- 1080 [27] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and André van der Hoek. 2018. On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by GitHub. *IEEE Transactions on Software Engineering* (2018), 1–1. <https://doi.org/10.1109/TSE.2018.2871083>
- 1085 [28] M. L. Guimarães and A. R. Silva. 2012. Improving early detection of software merge conflicts. In *2012 34th International Conference on Software Engineering (ICSE)*. 342–352. <https://doi.org/10.1109/ICSE.2012.6227180>
- [29] David Harris and Sarah Harris. 2007. *Digital Design and Computer Architecture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- 1090 [30] Sk Adnan Hassan, Zainab Aamir, Dongyoon Lee, James C. Davis, and Francisco Servant. 2023. Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1238–1255. <https://doi.org/10.1109/SP46215.2023.10179442>
- 1095 [31] Xianhao Jin. 2021. Reducing Cost in Continuous Integration with a Collection of Build Selection Approaches. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
- 1100 [32] Xianhao Jin and Francisco Servant. 2020. A Cost-efficient Approach to Building in Continuous Integration. In *International Conference on Software Engineering*.

- 1105 [33] Xianhao Jin and Francisco Servant. 2021. CIBench: A Dataset and Collection of Techniques for Build and Test Selection and Prioritization in Continuous Integration. In *International Conference on Software Engineering: Companion Proceedings*.
- [34] Xianhao Jin and Francisco Servant. 2021. What Helped, and What Did Not? An Evaluation of the Strategies to Improve Continuous Integration. In *International Conference on Software Engineering*.
- 1110 [35] Xianhao Jin and Francisco Servant. 2022. Which Builds are Really Safe to Skip? Maximizing Failure Observation for Build Selection in Continuous Integration. *Journal of Systems and Software* (2022).
- [36] Xianhao Jin and Francisco Servant. 2023. HybridCISave: A Combined Build and Test Selection Approach in Continuous Integration. *ACM Transactions on Software Engineering and Methodology* 32, 4, Article 93 (may 2023), 39 pages. <https://doi.org/10.1145/3576038>
- 1115 [37] B. K. Kasi and A. Sarma. 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In *2013 35th International Conference on Software Engineering (ICSE)*. 732–741. <https://doi.org/10.1109/ICSE.2013.6606619>
- 1120 [38] Ayaan M Kazerouni, James C Davis, Arinjoy Basak, Clifford A Shaffer, Francisco Servant, and Stephen H Edwards. 2021. Fast and Accurate Incremental Feedback for Students’ Software Tests using Selective Mutation Analysis. *Journal of Systems and Software* (2021).
- [39] Ayaan M. Kazerouni, Clifford A. Shaffer, Stephen H. Edwards, and Francisco Servant. 2019. Assessing Incremental Testing Practices and Their Impact on Project Outcomes.
- 1125 [40] Michele Lanza, Marco D’Ambros, Alberto Bacchelli, Lile Hattori, and Francesco Rigotti. 2013. Manhattan: Supporting real-time visual team activity awareness. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 207–210.
- 1130 [41] Olaf Leßenich, Sven Apel, and Christian Lengauer. 2015. Balancing Precision and Performance in Structured Merge. *Automated Software Engg.* 22, 3 (Sept. 2015), 367–397. <https://doi.org/10.1007/s10515-014-0151-5>
- [42] Olaf Leßenich, Janet Siegmund, Sven Apel, Christian Kästner, and Claus Hunsen. 2018. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering* 25, 2 (2018), 279–313.
- 1135 [43] Thomas W. MacFarland and Jan M. Yates. 2016. *Kruskal–Wallis H-Test for Oneway Analysis of Variance (ANOVA) by Ranks*. Springer International Publishing, Cham, 177–211. https://doi.org/10.1007/978-3-319-30634-6_6
- 1140

- [44] Chandra Maddila, Nachiappan Nagappan, Christian Bird, Georgios Gousios, and Arie van Deursen. 2021. ConE: A Concurrent Edit Detection Tool for Large Scale Software Development. *arXiv preprint arXiv:2101.06542* (2021).
- 1145 [45] M. Mahmoudi, S. Nadi, and N. Tsantalis. 2019. Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 151–162. <https://doi.org/10.1109/SANER.2019.8668012>
- 1150 [46] J.H. McDonald. 2014. *Handbook of Biological Statistics (3rd ed.)*. Sparky House Publishing, Baltimore, Maryland, 157–164.
- [47] T. Mens. 2002. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* 28, 5 (2002), 449–462. <https://doi.org/10.1109/TSE.2002.1000449>
- 1155 [48] Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In *International Conference on Automated Software Engineering*.
- [49] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. 2019. The life-cycle of merge conflicts: processes, barriers, and strategies. *Empirical Software Engineering* (02 2019). <https://doi.org/10.1007/s10664-018-9674-x>
- 1160 [50] Hoai Le Nguyen and Claudia-Lavinia Ignat. 2018. An Analysis of Merge Conflicts and Resolutions in Git-Based Open Source Projects. *Computer Supported Cooperative Work (CSCW)* 27, 3 (01 Dec 2018), 741–765. <https://doi.org/10.1007/s10606-018-9323-3>
- 1165 [51] Yuichi Nishimura and Katsuhisa Maruyama. 2016. Supporting Merge Conflict Resolution by Using Fine-Grained Code Change History. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* 1 (2016), 661–664.
- 1170 [52] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. 2019. Predicting Merge Conflicts in Collaborative Software Development. <https://arxiv.org/pdf/1907.06274.pdf>.
- [53] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu Lahiri, and Mike Kaufman. 2021. Can Program Synthesis Be Used to Learn Merge Conflict Resolutions? An Empirical Analysis. In *Proceedings of the 43rd International Conference on Software Engineering (Madrid, Spain) (ICSE '21)*. IEEE Press, 785–796. <https://doi.org/10.1109/ICSE43902.2021.00077>
- 1175

- 1180 [54] Karl Pearson. 1900. X. On the criterion that a given system of deviations
from the probable in the case of a correlated system of variables is such
that it can be reasonably supposed to have arisen from random sampling.
*The London, Edinburgh, and Dublin Philosophical Magazine and Journal
of Science* 50, 302 (July 1900), 157–175. [https://doi.org/10.1080/
14786440009463897](https://doi.org/10.1080/14786440009463897)
- 1185 [55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel,
M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A.
Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011.
Scikit-learn: Machine Learning in Python. *Journal of Machine Learning
Research* 12 (2011), 2825–2830.
- 1190 [56] Anita Sarma, David F Redmiles, and Andre Van Der Hoek. 2011. Palantir:
Early detection of development conflicts arising from parallel code changes.
IEEE Transactions on Software Engineering 38, 4 (2011), 889–908.
- [57] Francisco Servant. 2013. Supporting Bug Investigation using History Anal-
1195 ysis. In *International Conference on Automated Software Engineering*.
- [58] Francisco Servant and James A Jones. 2011. History Slicing. In *Interna-
tional Conference on Automated Software Engineering*. IEEE.
- [59] Francisco Servant and James A Jones. 2012. History Slicing: Assisting
Code-evolution Tasks. In *International Symposium on the Foundations of
1200 Software Engineering*.
- [60] Francisco Servant and James A Jones. 2012. WhoseFault: Automatic
Developer-to-Fault Assignment through Fault Localization. In *Interna-
tional Conference on Software Engineering*.
- [61] Francisco Servant and James A Jones. 2013. Chronos: Visualizing Slices of
1205 Source-code History. In *Working Conference on Software Visualization*.
- [62] Francisco Servant and James A Jones. 2017. Fuzzy Fine-grained Code-
history Analysis. In *International Conference on Software Engineering*.
- [63] Francisco Servant, James A Jones, and André Van Der Hoek. 2010. CASI:
preventing indirect conflicts through a live visualization. In *Proceedings of
1210 the 2010 ICSE Workshop on Cooperative and Human Aspects of Software
Engineering*. 39–46.
- [64] Bowen Shen, Muhammad Ali Gulzar, Fei He, and Na Meng. 2022. A
Characterization Study of Merge Conflicts in Java Projects. *ACM Trans.
Softw. Eng. Methodol.* (jun 2022). <https://doi.org/10.1145/3546944>
1215 Just Accepted.
- [65] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang
Wang. 2019. IntelliMerge: A Refactoring-Aware Software Merging Tech-
nique. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 170 (Oct. 2019),
28 pages. <https://doi.org/10.1145/3360596>

- 1220 [66] Marcelo Sousa, Isil Dillig, and Shuvendu Lahiri. 2018. Verified Three-Way Program Merge. In *Object-Oriented Programming, Systems, Languages & Applications Conference (OOPSLA 2018)*. ACM. <https://www.microsoft.com/en-us/research/publication/verified-three-way-program-merge/>
- 1225 [67] Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan, and Shuvendu K. Lahiri. 2022. Program Merge Conflict Resolution via Neural Transformers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Singapore, Singapore) (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 822–833. <https://doi.org/10.1145/3540250.3549163>
- 1230 [68] Gustavo Vale, Claus Hunsen, Eduardo Figueiredo, and Sven Apel. 2021. Challenges of Resolving Merge Conflicts: A Mining and Survey Study. *IEEE Transactions on Software Engineering* (2021), 1–1. <https://doi.org/10.1109/TSE.2021.3130098>
- 1235 [69] R. Yuzuki, H. Hata, and K. Matsumoto. 2015. How we resolve conflict: an empirical study of method-level conflict resolution. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*. 21–24. <https://doi.org/10.1109/SWAN.2015.7070484>
- 1240 [70] Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K. Lahiri. 2022. Using Pre-Trained Language Models to Resolve Textual and Semantic Merge Conflicts (Experience Paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual, South Korea) (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 77–88. <https://doi.org/10.1145/3533767.3534396>
- 1245 [71] Fengmin Zhu and Fei He. 2018. Conflict Resolution for Structured Merge via Version Space Algebra. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 166 (Oct. 2018), 25 pages. <https://doi.org/10.1145/3276536>
- 1250