Planning for Untangling: Predicting the Difficulty of Merge Conflicts

Caius Brindescu
Oregon State University
Corvallis, OR, USA
brindesc@oregonstate.edu

Iftekhar Ahmed
University of California, Irvine
Irvine, CA, USA
iftekha@uci.edu

Rafael Leano
Oregon State University
Corvallis, OR, USA
leanor@oregonstate.edu

Anita Sarma
Oregon State University
Corvallis, OR, USA
anita.sarma@oregonstate.edu

ABSTRACT

Merge conflicts are inevitable in collaborative software development and are disruptive. When they occur, developers have to stop their current work, understand the conflict and the surrounding code, and plan an appropriate resolution. However, not all conflicts are equally problematic—some can be easily fixed, while others might be complicated enough to need multiple people. Currently, there is not much support to help developers plan their conflict resolution. In this work, we aim to predict the difficulty of a merge conflict so as to help developers plan their conflict resolution. The ability to predict the difficulty of a merge conflict and to identify the underlying factors for its difficulty can help tool builders improve their conflict detection tools to prioritize and warn developers of difficult conflicts. In this work, we investigate the characteristics of difficult merge conflicts, and automatically classify them. We analyzed 6,380 conflicts across 128 java projects and found that merge conflict difficulty can be accurately predicted (AUC of 0.76) through machine learning algorithms, such as bagging.

CCS CONCEPTS

• **Software and its engineering** → *Software reliability*; *Software maintenance tools*; *Software design tradeoffs.*

KEYWORDS

Merge conflict difficulty prediction, Merge conflict resolution, Empirical analysis

ACM Reference Format:
Caius Brindescu, Iftekhar Ahmed, Rafael Leano, and Anita Sarma. 2020. Planning for Untangling: Predicting the Difficulty of Merge Conflicts. In *Proceedings of ACM conference (ICSE'2020).* ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Version Control Systems (VCS) have made large teams possible, enabling thousands of developers to contribute together in building Open Source Software (OSS), and proprietary software and technologies. However, despite the introduction of new, sophisticated, distributed version control systems, the basic protocol of using VCS still remains the same: code in private workspaces and synchronize periodically.

One challenge with this coordination protocol is merge conflicts. Merge conflicts occur when developers modify the same lines of code simultaneously. Research shows that merge conflicts are prevalent: about 19% of all merges end up in a merge conflict [2, 15, 37].

Merge conflicts have an impact on the code quality [2, 22, 50] and are disruptive to the development workflow. To resolve a merge conflict, a developer has to stop what they are doing and focus on the resolution. Resolving a conflict requires the developer to understand the conflicting changes and craft a solution that satisfies both sets of requirements driving the change. This disruption to the workflow can be compounded if conflict resolution requires additional expertise [21, 22, 50]. These factors can prompt developers to postpone the conflict resolution, or "kick the can" further down the road. In fact, a study by Nelson et al. [49] found that 56.18% of developers have deferred at least once when responding to a merge conflict. However, the later a conflict is resolved, the harder it is to recall the rationale of the changes. which makes the resolution process that much more difficult [7, 27]. As aptly put by a participant from the study by Nelson et al. [49]:

> *"Deferring a merge conflict simply kicks the can down the road (or off a cliff). Typically resolving the conflict only gets more difficult as time passes."*

However, sooner or later the conflict has to be resolved. To do so developers follow a process with four distinct resolution phases [49], as illustrated in Figure 1. Developers alternate between clean and conflicting states of code. Beginning from (1) the *development* stage, developers create an (2) *awareness* of conflicts within the codebase either passively when they face a conflict during a merge or by proactively monitoring ongoing changes. Once aware, developers begin (3) *planning* for a (4) *resolution* to fix the conflict. And finally, developers (5) *evaluate* the effectiveness of their deployed resolutions (returning to planning if the resolution fails).

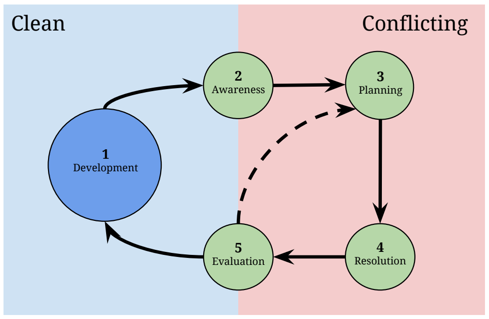

Figure 1: Model of Developer Processes for Managing Merge Conflicts, from Nelson et al. [49]

Several research works exist to support parts of the conflict resolution process. For the Development and Awareness phase, developers can benefit from workspace awareness tools [11, 15, 31, 53]. When working on the Resolution phase, developers can utilize different semi-automated merge techniques, such as unstructured merge [8, 12, 18, 47, 48], structured merge [16, 55, 60], semantics-based merge [10, 34], and hybrid merge [4, 5, 39]. The Evaluation phase has support through existing VCS (e.g., Git, SVN, TFS, CVS) and Continuous Integration systems (e.g., Jenkins, Travis CI, TeamCity).

None of these works, however, support the Planning phase of merge conflict resolution. We aim to close this gap and help developers plan their conflict resolution by predicting the difficulty of a merge conflict.

Our work can facilitate planning of conflict resolution in several ways. It can help developers plan: (1) when to resolve the conflict; if the conflict is simple they can resolve it instantaneously, otherwise they may need to allocate a longer resolution time period, (2) who to resolve it with; if a conflict is difficult they may need to coordinate a collaborative merge, (3) how much to review or test the merged code; if a conflict is difficult it may behoove the developers to more rigorously review and test the merged changes.

In this work, through a large scale empirical investigation we analyze what makes a merge conflict difficult, and whether we can predict the severity of a conflict from its underlying changes. More formally, we aim to answer the following research questions:

RQ1: How well can we predict the difficulty of merge conflicts?
RQ2: What makes a merge conflict difficult?
RQ3: How portable is our conflict difficulty prediction model?

To answer these research questions, we mine the characteristics of 6,380 merge conflicts from 128 Java projects in GitHub. To enable a prediction of merge conflict resolution, we gather a total of 16 process- and code-related metrics, such as the conflicting lines of code, differences in abstract syntax trees (AST), cyclomatic complexity (CC) etc. We use metrics that are available as the conflict develops (i.e., before the developer merges their changes), therefore, enabling developers lead time for their planning.

Our results show that we can predict the difficulty of merge conflicts accurately (an AUC of 0.76). Knowing which conflicts are difficult can help developers plan their conflict resolution. Our work also serves as a baseline prediction model for further research.

2 RELATED WORK

Merge conflicts are a common side effect of concurrent development [65]. While the use of version control systems flags divergent changes and prevents one change from overwriting another, they cannot always automatically resolve conflicts. Researchers have tried different approaches to help developers deal with conflicts: (1) early detection of conflicts, (2) merging assistance, and (3) prevention of conflicts.

2.1 Early detection

Conflicts tend to grow over time. Therefore, early detection helps limit the files and the size of the changes involved in a conflict. Workspace awareness tools monitor ongoing work to detect emerging conflicts. The goal of these tools is to "catch" the conflicts early so that it is easier to resolve them. Biehl et al. [11] propose FastDash, which identifies developers modifying the same file and notifies them about potential merge conflicts as they emerge. Hattori and Lanza [33] propose Syde, a tool that analyzes changes made to the source code at the level of AST operations. Syde detects conflicts by comparing the (AST) tree operations. Guimaraes and Silva [31] introduce WeCode, which continuously merges changes in the background to detect merge conflicts. Tools such as Palantir [53] and Crystal [15] proactively detect both merge conflicts, as well as semantic conflicts; the latter being conflicts that are revealed by failed builds or tests. Servant et al. [54] propose CASI, a tool that allows developers to visualize the code that their changes impact, with the aim of detecting semantic conflicts.

While these tools notify developers of emerging conflicts, they do not provide any assistance in resolving them. Our focus is on predicting the difficulty of merge conflicts so that developers can prioritize their resolution efforts.

2.2 Merging Assistance

Another major thread in merge conflict research is support for the resolution of merge conflicts. Mens [46] provides a comprehensive survey on the state of the art merging techniques. Apel et al. [4, 5] propose a new merging approach; called semi-structured merging. This technique considers the syntactical structure of the code that is to be merged. Lippe and van Oosterom [41] also propose a new merging technique, operational merging, which considers the changes that were done to the code, in addition to the end result.

While we do not attempt to provide resolution support to developers, our work may help developers choose one resolution strategy over other based on the difficulty of the conflict.

2.3 Prevention

Finally, another way to deal with conflicts is to prevent them from occurring in the first place. Kasi and Sarma [37] try to avoid merge conflicts altogether by scheduling tasks in a way that minimize the probability of conflict. Wloka et al. [61] introduce SafeCommit. It uses a static analysis approach to identify changes that can be committed safely, i.e. they do not cause any of the tests to fail. This allows developers to cherry pick the commits that are safe to commit (and avoid conflicts). Dewan and Hedge [24] propose a new development process that allows developers to synchronously collaborate on the conflicting code and solve the conflicts before finishing the task. Leßenich et al. [40] conducted a survey of 41 developers and inferred 7 indicators to predict the number of merge conflicts. Then they analyzed 163 open-source projects found that none of these 7 indicators suggested by the participating developer has a predictive power concerning the frequency of merge conflicts.

While some conflicts can be prevented, others are bound to occur. For example, Kasi and Sarma in their approach have to relax some constraints to allow some conflicts to occur when the space becomes too constrained. Leßenich et al. [39] and Cavalcanti et al. [17] examined 50 and 60 projects, respectively, to compare semi-structured and unstructured merge techniques in terms of how many conflicts they report. Both studies found that semi-structured merge techniques can reduce the number of conflicts by approximately half, but not eliminate them. Our work can be useful as a guide for which constraints (or conflicts) can be relaxed.

2.4 Conflict difficulty

Different studies have investigated ways to measure the amount of effort required to resolve a conflict. Resolution time varies significantly across projects and ranged not in hours, but in days [37]; and it can be used as a proxy to measure the difficulty of conflicts (difficult conflicts take more time to solve). The Orion approach by Prudencio et. al [51] tried to minimize the number of files to be *locked* using the actions applied in the file, and the committed actions. Their end goal was to *minimize* the number of conflicts that would occur, at the cost of reduced development concurrency. McKee et al. [44] and Nelson et al. [49] interviewed developers and then performed a follow-up survey with 162 developers to build a detailed understanding of developer perceptions regarding merge conflicts. They found, among other things, that complexity of the conflicting lines of code and file as a whole, number of LOC involved in the conflict, and developers' familiarity with the lines of code in conflict all impact how difficult developers find a conflict to resolve. Menezes et al. [45] used number of conflicting chunks to determine patterns that occur in merge conflicts.

All in all, none of these related works deals with the main purpose of the our work: the prediction of difficulty of potential merge conflicts in order to help developers prioritize merge conflicts to inspect, accomplish more things given tight schedule, and not waste reviewing effort on trivial merge conflict resolutions.

3 METHODOLOGY

To predict the difficulty of conflict resolutions, we collect a representative corpus of merge conflicts to be examined by four different machine learning classifiers. We use the following process during our study: (A) we collect a sample of Java projects hosted on GitHub; (B) we filter projects that do contain merge conflicts, are inactive, or toy projects; (C) we extract the relevant attributes needed for merge conflict analysis by conducting a literature survey; (D) we label a subset of merge conflicts manually; (E) we conduct supervised training with four machine learning classifiers; (F) we compare the results from each of the classifiers; (G) we perform feature selection; and, (H) we repeat steps (E) and (F) for cross-project merge conflict difficulty prediction. We describe each of these steps in further detail in the following subsections.

3.1 Project Selection Criteria

We made our project selections to be applicable to the requirements of the four machine learning classifiers and to be representative of code developed in the real world. Therefore, we only select active, open source projects from GitHub. We opted to select projects that use the same programming language to control for language-specific differences in the Lines of Code (LOC) metrics, program dependencies, and construct comparability. We use Java as our language of choice for two reasons: (1) Java is one of the most popular languages (according to the number of Java projects hosted on Github [30] and the Tiobe index [1]); and, (2) there are more mature analysis tools available for Java as compared to other programming languages.

We began by selecting 900 Java projects returned by GitHub search mechanism without any filtering criteria. From these 900 projects, we eliminate projects that were forked copies of other projects to prevent skewed results, leaving 500 projects in the end. Since some projects do not compile, either due to syntax, build errors, or missing dependencies, we eliminated an additional 300 projects. After filtering, our corpus contained 200 Java projects we were successfully able to compile and run.

We follow the guidelines presented by Kalliamvakou et al. [36] for mining Git repositories. We removed projects that were too small (fewer than 10 files, or fewer than 500 lines of code), and those that were not active in the past 6 months. We also removed projects that do not contain merge conflicts. These selection criteria were required since there is a long tail of small and short lived projects on GitHub; including trial projects, projects with a single author, or projects with no parallel development history, which did not have any merge commits. We focus on collaborative software development for this work, and we therefore remove projects that are not collaborative in nature. Our final corpus had 128 projects. Table 1 provides a summary of project statistics for these projects, including: number of lines, total number of commits, total number of merges, total number of merge conflicts, number of developers, and number of days that project has existed on GitHub as of March 1, 2018.

Table 1: Mined Projects Statistics

Dimension	Max	Min	Average	Std. dev.
Line count (LOC)	542,571	751	75,795.04	105,280.1
Total Commits	30,519	16	3,894.48	5,070.73
Total Merges	4,916	1	252.60	522.73
Total Conflicts	227	1	25.86	39.49
# Developers	105	4	72.76	83.19
Duration (days)	6,386	42	1,674.54	1,112.11

We also manually categorized the domain of the projects by looking at the project description and using the categories used by Souza et al. [23]. Table 2 has the summary of the domains of the 128 projects and their percentage of representation within our corpus.

3.2 Conflict Identification

We queried the repository of the 128 projects, from which we extracted 556,911 commits. This included 36,122 merge commits. Since

Table 2: Distribution of Projects by Domain

Domain	Percentage
Development	61.98%
System Administration	12.66%
Communications	6.42%
Business & Enterprise	8.10%
Home & Education	3.11%
Security & Utilities	2.61%
Games	3.08%
Audio & Video	2.04%

Git only stores information of commits, but does not record instances of merge conflicts we identified merge conflicts by following branch merges and analyzing the commits involved as shown in Figure 2. First, we identified merges as commits with two or more parents, such as commit *AB*. Then we merged the parents of that commit (A_n and B_n) using the `git merge` command. If the merge was unsuccessful then *AB* was marked as a merge conflict (m). Using this approach, we identified 6,380 merge conflicts.

We consider a merge conflict to be an instance when running `git merge` failed because of concurrent changes in the 2 (or more) branches being merged. A conflict can have multiple conflicting files, each with multiple conflicting chunks. For the purposes of this research, we focused on conflicts that occurred in source code files (`.java`).

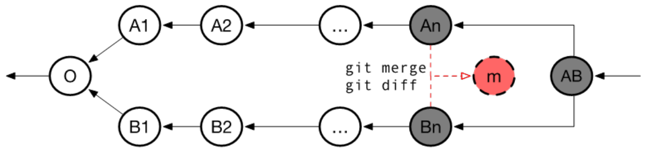

Figure 2: Identifying conflicts in git, merge AB has two parents A_n,B_n that cannot be merged automatically.

3.3 Data Collection

Once we identified a merge conflict, we extracted additional information relevant for the analysis of the conflict, such as: the authors involved in the commits, the commit message, the files that were edited, the changes that were made (by using the `git diff` command), and so on. This was done for the parent commits A_n and B_n, as well as all commits on either branch back to the base commit (i.e. the last shared parent commit). That is, from all commits A_1 to A_n and B_1 to B_n (in Figure 2).

Performance of any prediction is dependent on the features used. Therefore, we wanted to use a comprehensive set of features. In order to get an overview of the current state of the art research on merge conflict difficulty and metrics used, we conducted a literature survey. We targeted all full conference and journal papers related to merge conflict difficulty from 2008 to 2018 (inclusive) that appeared in top Software Engineering venues: ICSE, FSE, ASE, ICSM/ICSME, MSR, ESEM, TSE, TOSEM. Starting from the proceedings, we searched for a set of keywords including *"merge conflict difficulty," "merge conflict resolution effort"* etc. We found only 4 papers [44, 45, 49, 51] and we analyzed what metrics were used in the studies.

Based on the metrics reported in the identified papers and also based on intuition, we obtained 16 factors for each conflict, which we list in Table 3. We grouped these factors into five unique dimensions: size, complexity, diffusion, development pattern and comment [62]. We gathered these factors from either the Git repository, or we derived them by analyzing the source code, when the factors are related to the process and code metrics (characterized in numerical form).

Since certain metrics are calculated at different levels of granularity (e.g. complexity metrics are calculated at the method level), we aggregate all factors to a per conflict measurement using the average. In Table 3, the subscript is the operation used for aggregation. For example, for cyclomatic complexity CC_{sum} is the sum of the cyclomatic complexities of all the files modified in both branches. We calculated file-based metrics using all modified files and conflicting files. Size metrics use LOC as the unit of measurement.

We also include branch and author commits patterns. They refer to the temporal order in which commits are ordered in separate branches. The branch pattern reflects how commits are interleaved between branches. For example, in Figure 3, commit A_1 in branch *A* followed by commit B_1 in branch *B*, which yields the pattern *AB*. We continue building the pattern, until we reach the final *ABAABBBA*. We then collapse identical letters, yielding the final pattern *ABABA*. We collapse the letters, because we are interested in the interleaving, and not the total number of commits. A longer pattern means that the commits were more interleaved (tangled).

Similarly, author pattern shows how commits were interweaved between different authors. In Figure 3 we have 3 authors: John (J), Alice (A), and Oscar (O). Applying the same rules as the branch pattern, we end up with *JAOAJ*. Our rational for using these patterns as features in our analysis is that, the more interleaved (tangled) a development patterns is, the more difficult it should be to untangle it when resolving the merge conflict.

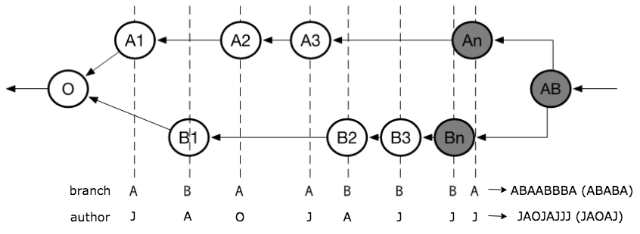
A1
A2
A3
An
O
AB
B1
B2
B3
Bn
branch A B A A B B B A → ABAABBBA (ABABA)
author J A O J A J J J → JAOJAJJJ (JAOAJ)

Figure 3: Example of calculating the branch and author patterns for a merge commit. Time flows from left to right and the arrows point to a commits parent(s).

3.4 Training Data Labeling

Before training the classifiers, we first manually labeled conflicts as either *severe* or *trivial* based on their difficulty of resolution. From the pool of 6,380 conflicts, we extracted and labeled a random sampling of 600 conflicts (approximately 10% of all conflicts). This random sample represents conflicts in 105 distinct projects out of the total of 128.

In order to validate our evaluation, the first two authors independently labeled 60 conflicts based on their difficulty. In order to

Table 3: Collected Process and Product Metrics

Category	Metric	Description	Source
Size	*Number of authors*	Number of different authors involved in all commits $A_1..A_n$, and $B_1..B_n$.	[49]
	Number of edits	The number of edits of each file	[45, 49]
	LOC	Size in Lines of Code (LOC) of commit A (LOC_A), B (LOC_B)	[45, 49]
	LOC_{diff}	LOC differences between commits A_n and B_n	[49]
Complexity	CC_{sum}	The total (sum) cyclomatic complexity of all files modified in both branches	[44, 49]
	CC_{max}	The maximum cyclomatic complexity of all files modified in both branches	[44, 49]
	CC_{avg}	The average cyclomatic complexity of all files modified in both branches	[44, 49]
Dev. Pattern	$Pattern_{branch}$	Length of commit pattern between branches	[51]
	$Pattern_{author}$	Length of author pattern between branches	[51]
Diffusion	$Files_{java}$	Total number of *java* files modified in branches A, B	[45]
	Files	Total number of files modified in branches A, B	[44, 49]
	$Dependency_{sum}$	Total dependencies (sum) of all files modified in both branches	[44, 49]
	$Dependency_{max}$	The maximum dependencies of all files modified in both branches	[44, 49]
	$Dependency_{avg}$	The average dependencies of all files modified in both branches	[44, 49]
	AST_{diff}	The number of differences of the AST trees between all modified files in both branches	[45]
Comment	*Comments*	All the comments from commits $A_1 \ldots A_n$, and $B_1 \ldots B_n$.	[45]

evaluate the difficulty, the authors recreated each conflict and attempted the resolution. The authors used the time required to solve the conflict as well as the cognitive load, in order to classify the merge conflict as difficult or *Severe* or *Trivial.* In order to validate the resolution, we compared our merge conflict resolution with the one that was checked in the version control system. We considered the developer's merge conflict resolution as the oracle, as it is most likely to be correct, given their in depth knowledge of the code. In all cases, our resolution was functionally equivalent to the developer's resolution. Listing 1 shows part of the author's resolution for a *Severe* merge conflict, which is functionally identical to the developers' resolution[1].

We checked for agreement by using Cohen's Kappa, and we achieved an IRR of 0.8. The two authors then independently coded the rest of the conflicts, in order to build the training set.

3.5 Feature Selection

To select the appropriate metrics we carry out an analysis of potential multicollinearity between the metrics. Previous research [6, 56] demonstrated that many process and source code metrics are correlated, both with each other, and with lines of code (LOC). Ignoring such correlations would lead to increased errors in the estimates of model performances, and increased standard errors of the predictions [32]. We checked for multicollinearity using the Variance Inflation Factor (VIF) [19] of each predictor in our model for our data set. VIF describes the correlation between predictors. A VIF score between 1 and 5 indicates moderate correlation with other factors, and these selected the predictors are with VIF < 5. This step was necessary since the presence of highly correlated factors forces the estimated regression coefficient of one variable to depend on other predictor variables that are included in the model. Out of the 16 factors, 5 had VIF ≥ 5, so we ended up using the remaining 11 factors for building the classifiers.

We further investigate the resulting eleven factors in their effectiveness in predicting the difficulty of a merge conflict, and report the results in Section 4. We find that prior work has also used a subset of these factors as a proxy for difficulty of a conflict [37, 51], which is encouraging for our own work.

3.6 Machine Learning

We trained and tested our sample using 4 different machine learning techniques: Support Vector Machines (SVM), Logistic Regression, Multi-Layer Perceptron (Perceptron), and Bayes Network (BayesNet). For all techniques, we used a 10-fold cross-validation on our sample of 600 labeled conflicts. We used a wide range of learning techniques to reduce the risk of dependence on a particular algorithm or implementation.

3.6.1 BayesNet. We use the Bayes Network algorithm as we expected features to not be independent and Bayesian Network does not have assumptions regarding independence. For example, LOC_{diff} and *files* (number of files) share a relation, as editing more files will also increase the LOC edited. Also, Bayesian Networks can represent richer models compared to naive Bayes classifiers. We used the SimpleEstimator to calculate the conditional probabilities used by the Bayes algorithm. Finally, BayesNet uses a hill-climbing greedy algorithm for evolving, combined with a K2 search algorithm to create its network. We configured with a batch size of 100.

3.6.2 Binomial Logistic Regression. For the Binomial Logistic Regression, we started with a full model using all 11 attributes from conflicts. This was followed by a model selection using *Akaike Information Criterion* (AIC), which estimates the information loss between models in comparison to the original. It ultimately selects

[1]https://github.com/neo4j/neo4j//commit/178be5

```
Factory<byte[]> xidGlobalIdFactory = createXidGlobalIdFactory();
... // no conflicts in these lines
txManager = new ReadOnlyTxManager( xaDataSourceManager, xidGlobalIdFactory, logging.
    getMessagesLog( ReadOnlyTxManager.class ) );
```

Listing 1: Authors' resolution of a *Severe* merge conflict. In this example, the developers made two concurrent refactorings to the `ReadOnlyTxManager` constructor, one of the refactorings introducing a new parameter.

the best model based on both the fit of the model and the information lost. We then use the selected attributes to build the new, final model, on which we evaluate.

3.6.3 Support Vector Machine (SVM). Based upon an assumption that conflicts would be linearly separated across factors, we selected SVM. Our SVM uses the standard *Radial Bases Function* (RBF) Kernel and for the other parameters we performed a grid search to choose the best classification found. This configuration result had a 1,000-cache size, a 1,000 size, a gamma (γ) of 0.0001, a C of 1,000 and one maximum iterator.

3.6.4 Multi-Layer Perceptron. We used a Multi-Layer Perceptron to see if it could leverage hidden relationships not explored in the other algorithms. We configured our Perceptron with a 0.3 learning rate, a 0.2 momentum, and 500 epochs. The Perceptron would terminate its validation testing after not being able to reduce its error 20 times in a row.

3.6.5 Bagging. We also used Auto-WEKA [38] for identifying the best classifier, which automatically searches through the joint space of WEKA's learning algorithms and their respective hyperparameter settings to maximize performance, using sequential model-based optimization [14] (a Bayesian optimization method). Though there is one Python based implementation called Auto-sklearn [26], we chose Auto-WEKA because it comprises a larger space of models and hyperparameters [38] compared to Auto-sklearn. This ended up identifying "Bagging (Bootstrap aggregating)" [13] as the best technique. "Bagging" is an ensemble based approach that uses multiple models to fit the bootstrap samples generated from the original data and then uses voting for classification. Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [20] was identified as the base learner (accessible at *weka.classifiers.rules.JRip*) by Auto-WEKA. These results were generated by running Auto-WEKA with random seed 123 for 4 hours.

3.7 Evaluation

We report the standard precision, recall, and AUC (Area Under the receiver operating characteristic Curve) to asses the performance of the prediction models, because it is independent of prior probabilities [9]. Also, AUC is a better measure of classifier performance than accuracy because it is not biased by the size of test data. Moreover, AUC provides a "broader" view of the performance of the classifier since both sensitivity and specificity for all threshold levels are incorporated in calculating AUC. Other work related to prediction have used AUC for comparison purposes [25, 28, 29, 63]. We list the formula used for calculating precision, recall and F-measure below. The AUC curve is created by plotting the recall against the false positive rate (FPR) at various threshold settings. We list the formula of FPR also.

- **Precision (P)**: A measure of whether the *Severe* predictions were *actually difficult.*

$$precision = \frac{t_p}{t_p + f_p} \tag{1}$$

- **Recall (R)**: A measure of the percentage of *Severe* instances that the approach managed to correctly predict.

$$recall = \frac{t_p}{t_p + f_n} \tag{2}$$

- **False positive rate (FPR)**: A measure of the ratio of the number of *Severe* conflict wrongly categorized and the total number of actual *Severe* conflicts.

$$FPR = \frac{f_p}{f_p + t_n} \tag{3}$$

3.8 Cross-project prediction

Cross-project prediction is important for some projects, specially projects that do not have historical data to perform any significant training. Hence, we investigated whether it is feasible to perform cross-project training following the method used by Rahman et al. [52]. We do so by training models on one project and testing on all other projects, ignoring time-ordering.

4 RESULTS

Here we discuss the results of our study by placing them in the context of three research questions, which investigate the the ability to predict the difficulty of a conflict (RQ1), factors that are useful in determining the difficulty of a conflict (RQ2), and whether we can perform cross-project merge conflict difficulty prediction (RQ3).

4.1 RQ1: How well can we predict the difficulty of merge conflicts?

To answer this research question, we trained four different machine learning algorithms: Bayes Network (BayesNet), Logistic Regression, Support Vector Machine (SVM), and Multi-Layer Perceptron (Perceptron). We also used Auto-WEKA [38], which automatically searches through the joint space of WEKA's learning algorithms and their respective hyperparameter settings to maximize performance and identify the best classifier. The algorithm with the best performance according to Table 4 is *"Bagging (Bootstrap aggregating)"* with *"RIPPER"* as the base learner, which has the highest AUC (0.85). Bagging is closely followed by Bayes Network, which still performs better-than-chance with 0.78 AUC. On the other hand,

Table 4: Performance of the classifiers. Bagging has the highest AUC at 0.85.

	Precision	Recall	AUC
SVM	0.70	0.70	0.56
L.R.[i]	0.70	0.65	0.73
Perceptron	0.75	0.69	0.75
Bayes Network	0.75	0.75	0.78
Bagging	0.79	0.79	0.85

[i] Logistic Regression

SVM performs worst (0.56 AUC). Table 4 shows the results in terms of precision, recall, and AUC.

Observation 1: The difficulty of resolving merge conflicts can be predicted with an AUC value of 0.85 when using a Bagging classifier.

Additionally, we use our Bagging model on the full corpus of 6,380 conflicts to see the characteristics of the merge conflicts; those that are predicted as *Severe* or *Trivial.* The model identifies 21% of the conflicts as *Severe,* and the rest 79% are classified as *Trivial.*

In our context, precision shows how well we correctly predict *Severe* conflicts, recall shows how many of the *Severe* conflicts we are able to find. We posit that in our scenario, precision has a higher priority than recall. This is because incurring more false positives is likely to make the developer to distrust the tool. As Bagging outperformed all other classifiers, we report the precision, recall, and AUC of Bagging separately for the two classes (*Severe* and *Trivial*) in Table 5.

Table 5: Results of the Bagging classifier, per class

Class	Precision	Recall	AUC
Severe	0.80	0.49	0.76
Trivial	0.79	0.94	0.85

Observation 2: Difficult merge conflicts can be predicted with a precision of 0.80 when using a Bagging classifier.

4.2 RQ2: Which characteristics of merge conflicts are associated with its difficulty?

In Section 4.1 we show that it is possible to predict the difficulty of a merge conflict with high accuracy. Our next step is to understand what are the characteristics of difficult conflicts. For this purpose we use *feature subset selection (FSS).* Specifically, we use "Wrapper" based methods, which considers the selection of a set of features as a search problem. Different combinations of features are prepared, evaluated and compared to other combinations [35]. "Wrapper" methods are also able to detect the possible interactions between features. In this technique, a predictive model is used to evaluate the combinations of features and a score is assigned based on accuracy of the model. We used RIPPER [20] as the predictive model for feature selection, since it was identified as best classifier for predicting merge conflict difficulty as explained in (Section 3.6).

Using FSS we obtain a set of ten factors from our initial set of 12. The ten selected factors encompass all the four metric categories (complexity, diffusion, size, and development pattern) to which the original factors belonged (see Table 3). All these factors can be calculated before the conflict occurs. This suggests that each of these categories are relevant in predicting difficult merge conflicts, even before the developer faces the conflict.

Table 6 presents additional information about these ten factors. Two of these factors include complexity metrics, such the CC_{sum} and the CC_{avg} referring to the mean and the sum of the cyclomatic complexities of all files modified in both branches. This suggests that the complexity of the code is an important metric that affects the difficulty in resolving a conflict. However, calculating complexity metrics require specialized (standalone) analysis tools. So, in the worst case, developers therefore have to "guestimate" the complexity of the code based on their own experience.

Table 6: Feature Selection Results (Sorted based on relative importance)

Category	Metric	FSS	Human Perceived Importance [44]
Complexity	CC_{sum}	1	1
Diffusion	$Dependency_{max}$	2	7
Diffusion	$Dependency_{avg}$	3	7
Complexity	CC_{avg}	4	1
Diffusion	AST_{diff}	5	6
Size	LOC_{diff}	6	6
Size	*LOC*	7	4
Size	*Number of authors*	8	Not mentioned
Dev. Pattern	$Pattern_{branch}$	9	Not mentioned
Dev. Pattern	$Pattern_{author}$	10	Not mentioned

Observation 3: We identify a subset of ten factors that include complexity, diffusion, size, and development pattern that can predict the difficulty of merge conflicts.

Table 6 also shows that development process related metrics are less influential (*Number of authors*, $Pattern_{branch}$ and $Pattern_{author}$) compared to code related metrics (CC_{sum}, $Dependency_{max}$, $Dependency_{avg}$, CC_{avg}, AST_{diff}, LOC_{diff} and *LOC*). This led us to investigate whether there is any difference between these two types of metrics when it comes to predicting merge conflict difficulty. We perform this analysis since both process and product metrics have known differences in prediction capability in the context of defect prediction [52]. We built the same set of classifiers shown in Table 4, once using only the process related metrics and once using only code related metrics. Tables 7 and 8 shows the results in terms of precision, recall, and AUC. Surprisingly, both process and code related metrics have similar prediction capabilities (AUC values of 0.69 vs. 0.70), unlike defect prediction, where process related metrics were found to be more powerful [52].

In the last column of Table 6 we report the factors that McKee at al. [44] identify as factors used by software practitioners to gauge merge conflict difficulty. They identified these factors through a

Table 7: Performance of the classifiers built using only process related metrics (*Number of authors*, $Pattern_{branch}$ and $Pattern_{author}$).

	Precision	Recall	AUC
SVM	0.68	0.69	0.55
L.R.[i]	0.69	0.63	0.70
Perceptron	0.65	0.63	0.72
Bayes Network	0.70	0.70	0.69
Bagging	0.69	0.71	0.69

[i] Logistic Regression

Table 8: Performance of the classifiers built using only code related metrics (CC_{sum}, $Dependency_{max}$, $Dependency_{avg}$, CC_{avg}, AST_{diff}, LOC_{diff} and *LOC*).

	Precision	Recall	AUC
SVM	0.65	0.68	0.52
L.R.[i]	0.68	0.69	0.67
Perceptron	0.55	0.48	0.49
Bayes Network	0.70	0.71	0.72
Bagging	0.70	0.71	0.70

[i] Logistic Regression

survey of software practitioners. Except for the complexity category, none of the other top features mentioned by practitioners are in the top features identified by FSS and vice versa. Clearly there is a disjoint between the human perceived features and machine learned features. We discuss this further in Section 5.

Observation 4: We identify differences between-human perceived features and machine-learned features for predicting the difficulty of merge conflicts, for example diffusion, which is perceived as unimportant by developers, but picked up as important by our model.

4.3 RQ3: Is cross-project training possible to predict difficult merge conflicts?

In RQ1 we tested our models using a 10-fold cross-validation with all our training data (600 conflicts). However, some projects may not have historical data to perform any significant training. Cross-project prediction has been investigated in other areas of software engineering such as defect prediction [42, 59, 64]. However, to the best of our knowledge, no one has investigated the applicability of cross-project merge conflict difficulty prediction. We followed the method used by Rahman et al. [52] to perform cross-project merge conflict difficulty prediction. We train models on one project using our best algorithm: "Bagging using RIPPER" and test on all other projects.

Figure 4 shows the portability of models across projects for different sets of evaluation metrics (precision, recall, F-measure and AUC). Performance clearly degrades in cross-project settings in comparison to "Bagging using RIPPER" algorithms performance of 0.85 AUC, shown in Table 4.

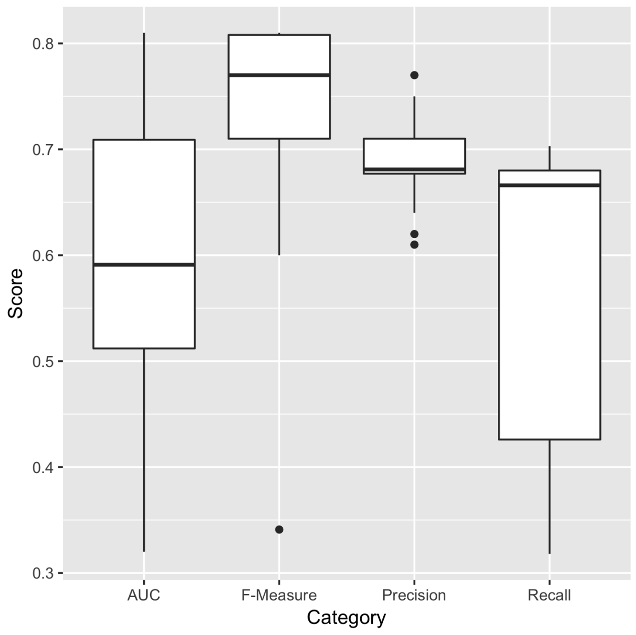

Figure 4: Precision, recall, f-measure and AUC for cross-project training.

Observation 5: Using cross-project training it is possible to build model that can be applied to an individual project with better-than-chance results (AUC=0.60 on average).

5 DISCUSSION

Centrality Matters: The goal of our study is to investigate whether it is feasible to predict the difficulty level of a merge conflict by using automated (machine learning) techniques. One factor that emerged as relevant is *Dependency*. A file that has high *Dependency* is likely to be highly coupled with other parts of the code, and therefore has high centrality. This is problematic for two reasons. First, as the file is central, it has more reasons to change. For example, a class with multiple functionalities (i.e. a God class [3]) is more likely to be changed for any kind of modification of the software. Second, the more a file gets changed, the more it's likely to be involved in a merge conflict. Moreover, the conflict is likely to contain disparate changes. This presents a challenge as the developer has to understand all the changes involved before resolving the merge conflict. Both our machine learning classifier and developers agree that this is a factor that determines the merge conflict resolution difficulty.

Tangled Changes: In our analysis we find that $Pattern_{author}$ is a significant factor in predicting merge conflict difficulty. A reason for this might be that the more authors that are involved in the development process, the more disparate the conflicting changes are—because each developer is likely working on a different functionality. So, whenever a conflict occurs, the developer has to understand the broader context of the changes before attempting to resolve the conflict.

Similarly, a longer branch pattern ($Pattern_{branch}$) means that the changes are more tightly tangled. This makes the changes more difficult to untangle when resolving the conflict. Interestingly, while our classifier identified these as important factors, the developers did not. This is an indication that developers are not aware that this could be a potential pain point.

Size does matter: Our classifier also identified the size difference (LOC_{diff}) between the two branches as a relevant factor for conflict difficulty prediction. This is intuitive, as the more lines are changed, the harder it is to understand the changes that were made (in that change set). This, in turn, makes it more difficult to the place that change set in the context of the rest of the code base. When dealing with the difference in terms of AST nodes (AST_{diff}), this becomes even more important, as the AST node difference is more likely to highlight semantic changes. In this case, both the classifier and the developers agree that the size difference between the two branches is an important factor in determining the merge conflict difficulty.

It's Complicated: It's well known that code with higher cyclomatic complexity is more difficult to understand. Therefore, it's not surprising that our classifier identified CC_{sum} as a significant factor for identifying the difficulty of a merge conflict. Another aspect is that code with high cyclomatic complexity is usually indicative of a complicated control structure. Therefore, conflicts in that area are more likely to be semantic in nature. Prior research has shown that areas of code affected by such conflicts are more likely to be buggy [2]. In this case, both the classifier and the developers agree that this is an important factor.

Learn from others: Our final observation is that models are portable between software projects. This is an indication that the factors that contribute toward merge conflict difficulty are more or less project independent. Therefore, our technique can be applied to new projects, or to projects with little development history and still prove beneficial to developers.

6 IMPLICATIONS

Our findings have multiple implications for tool builders, researchers, and practitioners.

6.1 Researchers

Our model for predicting merge conflict difficulty achieved an AUC value of 0.76 when predicting the minority class, which is a high value compared to a baseline of random classification [43]. However, there is still room for improvement. The research community should focus on identifying different types of factors (social, product and process) and investigate their effect on overall prediction accuracy, with the goal of improving the overall prediction accuracy.

Our results inform future research by providing insights into the factors that are associated with the difficulty of a merge conflict. Projects share similar features, as demonstrated by the moderate performance of cross-project merge conflict difficulty prediction. This also indicates that the prediction process can be bootstrapped even for project that lacks history. We recommend that researchers should also focus on identifying the best project selection criteria for bootstrapping cross-project prediction. For bootstrapping, we should use similar projects. However, what constitutes as a similarity metric between the project, in this context, is still an open research question. Our results show that cyclomatic complexity is the most important metric when predicting difficulty. (Table 6), so projects with similar Cyclomatic complexity should could used for bootstrapping. However, further investigation is required to make any conclusive or definitive remarks.

We also found that Auto-WEKA, which automatically searches through the joint space of WEKA's learning algorithms, and their respective hyperparameters, helped us to identify the best classifier and increased the AUC value from 0.78 to 0.85 (Table 4). Our finding is inline with the findings of other researchers [57, 58] who have shown the benefit of parameter optimization in improving classifier performance. Therefore, we recommend that researchers using machine learning classifiers should seriously consider parameter optimization to ensure the best performance of the classifiers.

6.2 Tool builders

When looking at the types of factors that make a merge conflict difficult, we identified categories relating to the complexity of the code (*Complexity*), extent of the change (*Diffusion*) and the length of branch pattern (*Development Pattern*). The *Complexity* and *Diffusion* metrics are already used by researchers and tool builders for merge conflict prediction. However, we are the first to associate the length of a branch pattern to merge conflicts and their difficulty. Further research can help identify threshold of branch pattern after which a merge conflict becomes severe. Tool builders can use such thresholds as a criteria to filter and prioritize the notifications about potential conflicts. This will not only help users manage the information load, but also will have impact on the quality of the final product [2].

6.3 Developers

Our results indicate that the more *"tangled"* a piece of code is, the more difficult it will be to resolve a conflict related to that code. So developers can use the length of $Pattern_{author}$ and $Pattern_{branch}$ in deciding merge conflict resolution strategy. Moreover, it's more likely that a code with bigger pattern length has diverged a lot from the initial point and has become difficult for any individual to understand completely. In such cases, it will be more productive to do a collaborative merge [21], instead of a developer performing the merge by herself.

We also found that all ten significant factors in determining merge conflict difficulty can be collected even before the merge conflict actually occurs. Current awareness tools are already collecting these information when predicting emerging conflicts. Therefore, without further overhead, awareness tools can use our model to predict the difficulty of the emerging conflict, which can be then used as a prioritization criteria when notifying users. Tools can also recommend developers who are most suited to resolve a conflict based on the historical data of merge resolutions. The rationale would be that developers with more experience of resolving difficult conflicts in the past are suitable candidates for resolving a *Severe* conflict, as compared to someone who lacks the experience.

7 THREATS TO VALIDITY

Our research findings may be subject to the concerns that we list below. We have taken all possible steps to neutralize the impacts of these possible threats, but some couldn't be mitigated and it's possible that our mitigation strategies may not have been effective.

Our samples have been from a single source - Github. This may be a source of bias, and our findings may be limited to open source programs from Github. However, we believe that the large number of projects sampled more than adequately addresses this concern.

Another threat to our findings is that, Git tracks the version history of a project as it occurred but it also allows history rewriting using the "rebase" command. It is known that some development teams use "rebase" instead of "merge" to reintegrate branches [18] which means that may have missed merges in our analysis and the number of merges we analyzed is a lower bound as compared to the actual total number of merges in the projects.

Although we use 24 factors spanning across six categories, there are likely other features that we did not measure. For example, we suspect that the design patterns of a program might influence the likelihood of a conflict resolution being difficult. We plan to expand our metric set to include additional categories in future work.

Our training and testing data had to be manually labeled since this information is not currently available in CM systems or issue trackers. Therefore, our labels may not accurately represent the real merging difficulty because of lack of domain expertise. Our labeling process included inter-rater reliability to prevent individual bias and to reduce this threat. Additionally, the experience and familiarity with the source code and the project can make a conflict difficult to resolve for one developer but simple for another. As we had multiple researchers and we also had a high inter-rater agreement, we assume this should minimize the aforementioned threat.

Another threat would be that we excluded non-source files from our manual analysis (e.g. configuration xml file etc.), but changes to non-source files can have impact on the program's execution if these files are involved in the build/deploy process or for code generation and ultimately make the merge resolution difficult.

8 CONCLUSIONS AND FUTURE WORK

In this empirical study, the first of its kind, we investigated the different aspects that can impact the difficulty level of resolving merge conflicts. We evaluated five different classification techniques from different families and identified "Bagging using RIPPER" as the base learner to be the best model; with an AUC of 0.76. We also identified a set of ten metrics that are most influential while predicting the difficulty level of a conflict which include metrics about the complexity of the code, the size of the change, and development pattern etc. We also found that there is a disconnect between the factors developers use to gauge the difficulty of a conflict and the factors our automatic classification technique identified as important. Finally, we showed that we are able to perform cross-project merge conflict difficulty prediction; with median AUC of 0.60. Therefore, our results show that we can bootstrap prediction in projects with no (labeled) data or only small amount of history, by training on other projects. Our study opens a new avenue in Software Engineering research related to predicting the difficulty level of a merge conflict and help developers plan the merge conflict management process efficiently.

We also provide actionable implications for researchers, tool builders, and practitioners to harness the results of our study. In future work, we hope to explore whether these factors can be seamlessly merged into tools or techniques to assist developers' workflows.

REFERENCES

[1] 2017. TIOBE Index for April 2017. https://tiobe.com/tiobe-index//. Accessed: 2017-04-18.
[2] Iftekhar Ahmed, Caius Brindescu, Umme Ayda Mannan, Carlos Jensen, and Anita Sarma. 2017. An Empirical Examination of the Relationship between Code Smells and Merge Conflicts. In *Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium on*. IEEE, 58–67.
[3] Iftekhar Ahmed, Umme Ayda Mannan, Rahul Gopinath, and Carlos Jensen. 2015. An empirical study of design degradation: How software projects get worse over time. In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*. IEEE, 1–10.
[4] 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*. ACM, 120–129.
[5] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 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*. ACM, 190–200.
[6] Henrike Barkmann, Rüdiger Lincke, and Welf Löwe. 2009. Quantitative evaluation of software quality metrics in open-source projects. In *2009 International Conference on Advanced Information Networking and Applications Workshops*. IEEE, 1067–1072.
[7] Stephen P Berczuk and Brad Appleton. 2002. *Software configuration management patterns: effective teamwork, practical integration*. Addison-Wesley Longman Publishing Co., Inc.
[8] Thomas Berlage and Andreas Genau. 1993. A framework for shared applications with a replicated architecture. In *ACM Symposium on User Interface Software and Technology*. Citeseer, 249–257.
[9] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. 2007. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, 11–18.
[10] Valdis Berzins. 1994. Software merge: semantics of combining changes to programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 6 (1994), 1875–1903.
[11] Jacob T Biehl, 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*. ACM, 1313–1322.
[12] Barry W Boehm, John R Brown, and Mlity Lipow. 1976. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 592–605.
[13] Leo Breiman. 1996. Bagging predictors. *Machine learning* 24, 2 (1996), 123–140.
[14] Eric Brochu, Vlad M Cora, and Nando De Freitas. 2010. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599* (2010).
[15] 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*. ACM, 168–178.
[16] Jim Buffenbarger. 1995. Syntactic software merging. In *Software Configuration Management*. Springer, 153–172.
[17] Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. 2015. Assessing semistructured merge in version control systems: A replicated experiment. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–10.
[18] Scott Chacon and Ben Straub. 2014. *Pro git*. Apress.
[19] Jacob Cohen, Patricia Cohen, Stephen G West, and Leona S Aiken. 2013. *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge.
[20] William W. Cohen. 1995. Fast Effective Rule Induction. In *Twelfth International Conference on Machine Learning*. Morgan Kaufmann, 115–123.
[21] Catarina Costa, Jair Figueiredo, Anita Sarma, and Leonardo Murta. 2016. TIPMerge: recommending developers for merging branches. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 998–1002.
[22] Cleidson RB De Souza, David Redmiles, and Paul Dourish. 2003. Breaking the code, moving between private and public work in collaborative software development. In *Proceedings of the 2003 International ACM SIGGROUP conference on Supporting group work*. ACM, 105–114.
[23] Lucas Batista Leite De Souza and Marcelo De Almeida Maia. 2013. Do software categories impact coupling metrics?. In *Proceedings of the 10th working conference on mining software repositories*. IEEE Press, 217–220.

[24] Prasun Dewan and Rajesh Hegde. 2007. Semi-synchronous conflict detection and resolution in asynchronous software development. In *ECSCW 2007*. Springer, 159–178.

[25] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. 2018. A developer centered bug prediction model. *IEEE Transactions on Software Engineering* 44, 1 (2018), 5–24.

[26] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Advances in neural information processing systems*. 2962–2970.

[27] M. Fowler. [n.d.]. Continuous Integration. http://martinfowler.com/articles/continuousIntegration.html. Accessed: 2017-03-1.

[28] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. 2012. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 171–180.

[29] Emanuel Giger, Martin Pinzger, and Harald C Gall. 2011. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 83–92.

[30] GitHub Inc. [n.d.]. Software Repository. http://www.github.com.

[31] Mário Luís Guimarães and António Rito Silva. 2012. Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 342–352.

[32] Frank E Harrell Jr. 2015. *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. Springer.

[33] Lile Hattori and Michele Lanza. 2010. Syde: a tool for collaborative software development. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 235–238.

[34] Daniel Jackson, David A Ladd, et al. 1994. Semantic Diff: A Tool for Summarizing the Effects of Modifications.. In *ICSM*, Vol. 94. 243–252.

[35] George H John, Ron Kohavi, and Karl Pfleger. 1994. Irrelevant features and the subset selection problem. In *Machine Learning Proceedings 1994*. Elsevier, 121–129.

[36] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 92–101.

[37] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 732–741.

[38] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. 2017. Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *The Journal of Machine Learning Research* 18, 1 (2017), 826–830.

[39] Olaf Leßenich, Sven Apel, and Christian Lengauer. 2015. Balancing precision and performance in structured merge. *Automated Software Engineering* 22, 3 (2015), 367–397.

[40] 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.

[41] Ernst Lippe and Norbert Van Oosterom. 1992. Operation-based merging. In *ACM SIGSOFT Software Engineering Notes*, Vol. 17. ACM, 78–87.

[42] Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. 2012. Transfer learning for cross-company software defect prediction. *Information and Software Technology* 54, 3 (2012), 248–256.

[43] Simon Mason and N.E. Graham. 2002. Areas beneath the relative operating characteristics (ROC) and relative operating levels (ROL) curves: Statistical signié cance and interpretation. *Quarterly Journal of the Royal Meteorological Society* 128 (07 2002), 2145 – 2166. https://doi.org/10.1256/003590002320603584

[44] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. 2017. Software practitioner perspectives on merge conflicts and resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 467–478.

[45] Gleiph Ghiotto Lima Menezes, Leonardo Gresta Paulino Murta, Marcio Oliveira Barros, and Andre 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).

[46] Tom Mens. 2002. A state-of-the-art survey on software merging. *IEEE transactions on software engineering* 28, 5 (2002), 449–462.

[47] Webb Miller and Eugene W Myers. 1985. A file comparison program. *Software: Practice and Experience* 15, 11 (1985), 1025–1040.

[48] Eugene W Myers. 1986. AnO (ND) difference algorithm and its variations. *Algorithmica* 1, 1-4 (1986), 251–266.

[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* (2019), 1–44.

[50] Antti Nieminen. 2012. Real-time collaborative resolving of merge conflicts. In *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*. IEEE, 540–543.

[51] João Gustavo Prudêncio, Leonardo Murta, Cláudia Werner, and Rafael Cepêda. 2012. To lock, or not to lock: That is the question. *Journal of Systems and Software* 85, 2 (2012), 277–289.

[52] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 432–441.

[53] Anita Sarma, David F Redmiles, and Andre Van Der Hoek. 2012. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering* 38, 4 (2012), 889–908.

[54] Francisco Servant, James A Jones, and André Van Der Hoek. 2010. CASI: preventing indirect conflicts through a live visualization. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 39–46.

[55] Haifeng Shen and Chengzheng Sun. 2005. Syntax-based reconciliation for asynchronous collaborative writing. In *2005 International Conference on Collaborative Computing: Networking, Applications and Worksharing*. IEEE, 10–pp.

[56] Martin Shepperd. 1988. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal* 3, 2 (1988), 30–36.

[57] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 321–332.

[58] Ayse Tosun and Ayse Bener. 2009. Reducing false alarms in software defect prediction by decision threshold optimization. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 477–480.

[59] Burak Turhan, Ayse Tosun, and Ayse Bener. 2011. Empirical evaluation of mixed-project defect prediction models. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 396–403.

[60] Bernhard Westfechtel. 1991. Structure-oriented merging of revisions of software documents. In *Software Configuration Management Workshop: Proceedings of the 3 rd international workshop on Software configuration management*, Vol. 12. 68–79.

[61] Jan Wloka, Barbara Ryder, Frank Tip, and Xiaoxia Ren. 2009. Safe-commit analysis to facilitate team software development. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 507–517.

[62] Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. 2016. Predicting crashing releases of mobile applications. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 29.

[63] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2014. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 182–191.

[64] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 91–100.

[65] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 429–445.