Predicting Textual Merge Conflicts

by

Moein Owhadi Kareshk

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

During collaborative software development, developers often use branches to add features or fix bugs. When merging changes from two branches, conflicts may occur if the changes are inconsistent. Developers need to resolve these conflicts before completing the merge, which is an error-prone and time-consuming process. Early detection of merge conflicts, which warns developers about resolving conflicts before they become large and complicated, is among the ways of dealing with this problem.

Existing techniques do this by continuously pulling and merging all combinations of branches in the background to notify developers as soon as a conflict occurs, which is a computationally expensive process. One potential way for reducing this cost is to use a machine learning based conflict predictor that filters out the merge scenarios that are not likely to have conflicts, *i.e. safe merge scenarios*. In this thesis, we assess if conflict prediction is feasible. We employed binary classifiers to predict merge conflicts based on 9 light-weight Git feature sets. We train and test predictors for each repository separately.

To evaluate our predictors, we perform a large-scale study on 147,967 merges from 105 GitHub repositories in seven programming languages. Our results show that decision trees can achieve high f1-scores, varying from 0.93 to 0.95 for repositories in different programming languages when predicting safe merges. The f1-score is between 0.45 and 0.71 for the conflicting merges. Our results indicate that predicting conflicts is feasible, which suggests it may successfully be used as a pre-filtering criteria for speculative merging.

Preface

Chapter 4 of this thesis has been published as M. Owhadi Kareshk, S. Nadi, "Scalable Software Merging Studies with MergAnser," Proceedings of the 16^{th} International Conference on Mining Software Repositories (MSR), 2019 [57].

Chapters 5, 6 and 7 of this thesis has been published as M. Owhadi Kareshk, S. Nadi, J. Rubin, "Predicting Merge Conflicts in Collaborative Software Development," Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2019 [58].

This project has been partially funded through a 2017 Samsung Global Research Outreach (GRO) program.

To my teachers

Acknowledgements

I would like to thank my supervisor, Dr. Sarah Nadi, for her helps and support during the time I was working with her. This thesis would not have been possible without her constructive feedback.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern software systems are commonly built by large and distributed teams of developers. Thus, improving the collaborative software development experience is important. Distributed Version Control Systems (VCSs), such as Git, and social coding platforms, such as GitHub, have made such collaborative software development easier. However, despite its advantages, collaborative software development also gives rise to several issues [13], [38], including merging and integration problems[6], [51].

Branching and merging are among the most important Git features that provide systematic support for collaborative development. These features allow developers to have multiple copies of the code that are called *branches* in Git to be able to work simultaneously. This isolates the changes and prevents breaking one copy of the code because of changes that other developers made. When a branch is ready to be merged into another branch of the code, Git helps developers to perform it using a one-line command. As we show in Fig. 1.1, each merge scenario has four main components. *Ancestor* is the starting point of branching and *parents* are the last *commits* on each branch. Git stores the development history as commits, which are the group of code changes with relevant information such as a message to describe the changes. Finally, *merge commit* is the result of merging which is automatically created by Git.

However, when two developers change the same part of the code, Git cannot decide which change to choose and reports *textual conflicts*. In this situation,

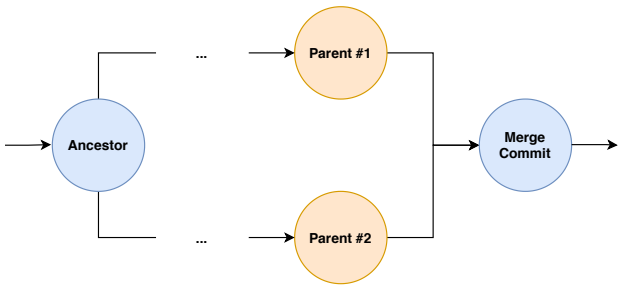

Figure 1.1: A merge Scenario in Git

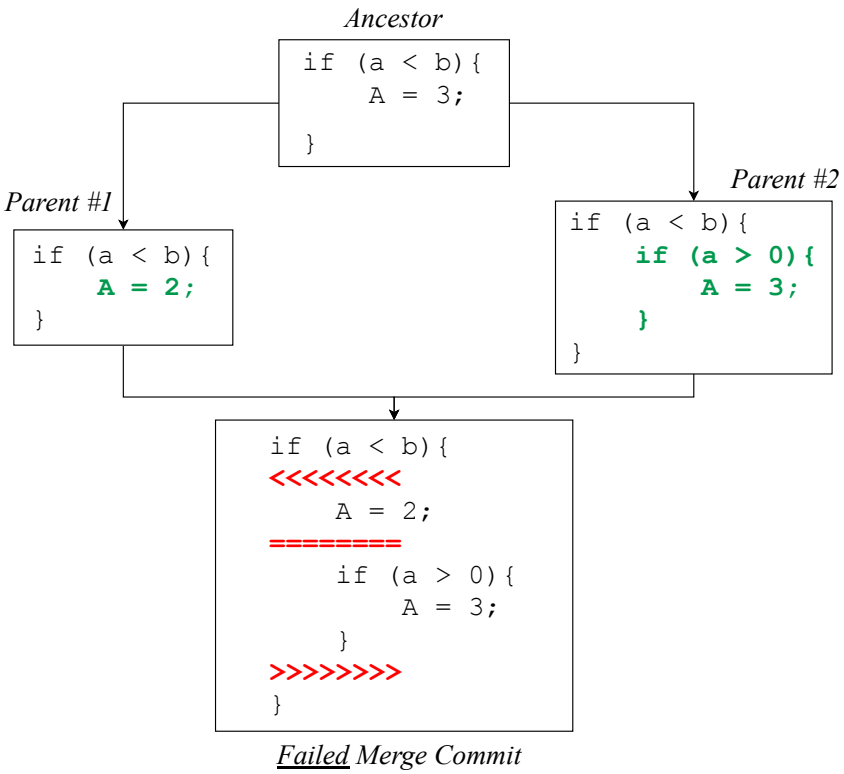

Figure 1.2: A simple conflicting merge scenario

the developers need to resolve the conflict manually, which is an error-prone and time-consuming task that wastes resources [14], [64]. Fig 1.2 shows a simple textual conflict, reported by Git with special tags. In this scenario, Git cannot decide between editing an assignment and adding a condition (changes are shown in green). Hence, it reports a conflict with special tags (in red) and asks developers to manually resolve it.

Working on the same code for a long time without proper communication between developers increases the change of having an unintentional conflict. Hence, previous research proposed new techniques and tools to resolve conflicts as soon as they occur [17], [32], [64]. *Speculative merging* [11], [16], [32],

[40] is one of the most effective ways for reducing conflicts by continuously merging the branches in the background and reporting any conflict as soon as possible, preferably, before the time developers need to merge the code. Git uses unstructured merge which is very fast. However, speculative merging might be computationally expensive due to a large number of branches and the number of combinations of branches that need to be merged. One possible solution to reduce the computational cost of speculative merging is filtering-out the merge scenarios with a low probability of having conflicts. Therefore, given a merge scenario, it is useful to be able to predict the chance of conflicts.

To the best of our knowledge, there have been two attempts at predicting merge conflicts in the past [7], [45]. The first study [45] looked for correlations between various features and merge conflicts and found that none of the features have a strong correlation with merge conflicts. The authors concluded that merge conflict prediction may not be possible. However, we argue that lacking correlation does not necessarily preclude a successful classifier, especially since the study did not consider the fact that the frequency of conflicts is low in practice and most of the standard forms of statistics and machine learning techniques cannot handle imbalanced data well. The second study [7] investigates the relationship between two types of code changes, edits to the same method and edits to dependent methods, and merge conflicts. The authors report recall of 82.67% and precision of 57.99% based on counting how often a merge scenario that had the given change was conflicting. This means that this second study does not build a prediction model that is trained on one set of data and evaluated on unseen merge scenarios.

Since neither of the above works built a prediction model that is suitable for imbalanced data and has been evaluated on unseen data, it is still not clear if predicting merge conflicts is feasible in practice, especially while using features that are not computationally expensive to extract. Additionally, both papers focus only on Java repositories and do not consider other programming languages. In this thesis, we investigate if predicting merge conflicts using inexpensive-to-extract Git features is feasible in practice.

1.1 Research Questions

In this research work, we aim to answer the following research questions:

RQ1: Which characteristics of merge scenarios have more impact on conflicts? We aim to investigate which features in Git history have an impact on conflicts. Our focus is the features that are not expensive to extract. Moreover, we want to see if this impact varies in repositories written in different programming languages.

RQ2: Are merge conflicts predictable using only Git features? We use machine learning techniques to implement conflict predictor models and evaluate them to see if predicting merge conflict is possible in practice.

1.2 Study Overview

To answer the above-mentioned questions, we study 105 well-engineered repositories that are listed in the reaper dataset [54], and that are written in seven different programming languages (C, C#, C++, Java, PHP, Python, and Ruby). We collect 147,967 merge scenarios from these repositories and (1) analyze the correlation between the features and conflicts and (2) train a separate binary classifier for each repository to predict conflicting merge scenarios.

To train our classifiers, we use a total of 9 feature sets that can be extracted solely through Git commands. We intentionally use only features that can be extracted from version control history so that our prediction process can be efficient (*e.g.*, as opposed to features that may require code analysis). Furthermore, we use cost-sensitive decision trees [60] and random forests [46], to take into account the specific characteristics of merge data, such as being imbalanced, in our classifiers. Figure 1.3 shows an overview of our methodology, which consists of three stages, as follows:

1. ***Collecting Merge Scenarios (Chapter 4)*** As the first step, we need to collect merge scenarios. We do so by mining the Git history of the target repositories.

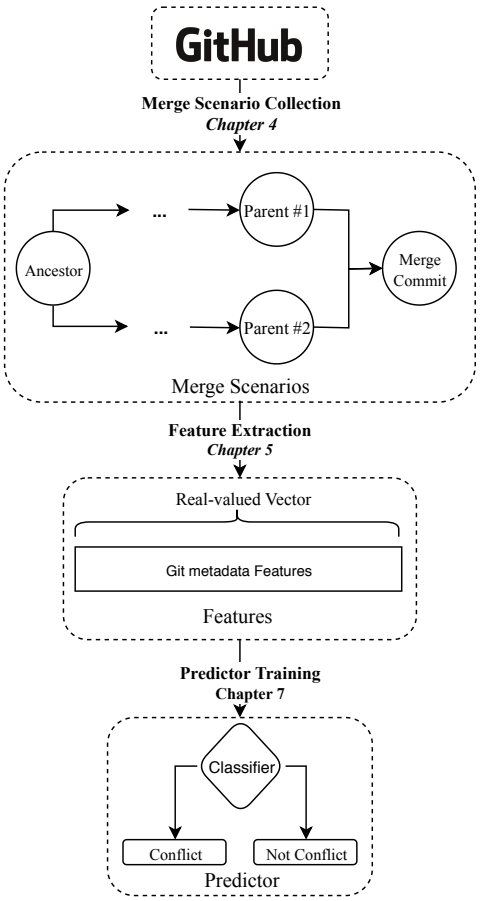

Figure 1.3: The Methodology for Creating the Proposed Conflict Prediction

2. ***Feature Extraction (Chapter 5):*** In the second stage, we extract the features that we will later use to build the prediction model. Using Git history, we extract features from both branches being merged.

3. ***Prediction (Chapter 7):*** In the last stage, we use statistical machine learning techniques to build a prediction model. Specifically, we use a binary classifier that aims to classify conflicting and not conflicting (safe) merge scenarios. Since conflicts happen in only a few numbers of merge scenarios, the classifier should be capable of handling imbalanced data. We employ imbalanced data handling techniques to train tree-based ensemble models.

1.3 Thesis Organization

We provide an overview of the background of this thesis in the next chapter. After that, we briefly introduce the related work to this thesis. We introduce our data gathering tool, MergAnser, and our dataset in Chapter 4. We then introduce the features we extracted from our data and the intuition behind using each of them in Chapter 5. Finally, we aim to answer our research questions in Chapter 6 and Chapter 7, respectively.

1.4 Thesis Contributions

The contributions of this thesis are:

1. We build a scalable tool (MergAnser) to extract the data of Git history as a normalized SQL database.
2. We create a set of potentially predictive features for merge conflicts based on the literature on software merging.
3. We apply systematic statistical machine learning strategies for handling the imbalanced data of software merging.
4. We design effective machine learning classifiers for textual conflicts in seven programming languages.
5. We evaluate our models using $147,967$ merge scenarios extracted from 105 well-engineered GitHub repositories written in different programming languages.

Chapter 2

Background

We describe the background of our research in this chapter. We first describe collaborative software development and Git, as a widely used version control system. We also introduce speculative merging which helps developers to reduce the number of merge conflicts in software merging. After that, we introduce the machine learning approaches we use for predicting merge conflicts.

2.1 Collaborative Software Development with Git

Today's software industry relies on collaborative development. Developers need to work on the same code at the same time and managers should be able to manage the team and have a clear view of the developers' progress. There are different tools that help software teams to manage their collaboration such as online messaging and version control systems.

Git is a vastly used version control system that helps developers store the development history, share software artifacts, and collaborate with each other [13], [38], [39]. Using Git, developers can work on the same code-base simultaneously. Git is a *distributed* version control system, which means that there is no central server that stores the main version separately and local versions do not have any advantage or priority in comparison with each other. We provide an overview of Git concepts that are most relevant to this thesis in the following.

Repository Each software project is called a *repository* in Git. A repository can contain textual (*e.g.* code, documentation, and configurations) and binary (*e.g.* image, video, and executable) files. While Git can store the binary files, only the changes in textual files are tracked.

Commits Developers specify all related changes that are made in the code for a specific purpose as a commit. Each commit has a SHA-1 hash which is its unique identifier. Moreover, each commit has a *commit message* that is a textual explanation about the code changes and their intentions in that commit. The commit messages should be expressive so that developers can have a global overview of the development history by reading commit messages. A commit can contain multiple changes such as adding or removing code snippets and renaming or deleting files. Each commit has a pointer to its *parent*, which is the previous commit that the current commit is based on. Merge commits are special cases and have more than one parent as we explain later.

Branches When multiple developers work on the same repository, they need to keep at least one copy of the code safe and reliable when adding new features or fixing bugs. Therefore, Git allows developers to have multiple copies of their code, each for a different purpose. These copies are called *branches*, and each branch has a unique name. The initial and default name of the first branch in Git is *master*. Branching is a key Git feature that allows developers to work in parallel.

Merging Developers *merge* the new branches with all of their changes to the base branch when they finish the development of the new branches and test them successfully. This way of merging is so-called *direct merging*. Git merge uses *unstructured merging*, which is line-based and does not consider the structure of the code. While this allows Git to be language-independent, it may result in failed merging. Git supports n-way merging (also called *Octopus Merging*), which allows developers to merge more than two parents into a

branch. However, since this is not a common way of merging in Git, we focus on 3-way merging, where there are only two parents to merge.

As we show in Fig 1.1, there are four important commits involved in each merge scenario:

1. **Ancestor:** The commit that branching started from.
2. **Parent #1:** The last commit in the branch that we want to merge into.
3. **Parent #2:** The last commit in the branch that we want to merge from.
4. **Merge Commit:** An automatically-generated commit by Git and it is the result of merging of Parent #1 and Parent #2.

Merge Conflicts When the same line of the code is changed in two branches, Git cannot decide which change to choose and therefore reports a *merge conflict* with specific tags. Git allows developers to edit the conflicting code snippets which are separated by tags by choosing either of the parents or combining the code snippets. Developers typically resolve these conflicts manually, which is an error-prone and time-consuming task [14], [64].

Rebasing There are two main ways of integration in Git, merging and rebasing. While merging keeps the history of branching in history, rebasing applies the commits of a branch on top of the other one and therefore, does not stores the branching history. Rebasing makes the Git history clean (by not generating merge commits) and linear but prevents us to analyze the merging in development history.

GitHub is an online social coding platform that operates based on Git and offers an additional set of features via a web-based user interface. This service allows organizations, teams, and developers to have their profiles and follow each other's work. Each profile has its Git repositories which can be either public or private. Repositories also can have wiki pages to provide additional information or documentation. To work on a project separately, developers

can copy a project with all of its content to their profile which is referred to as *forking* on GitHub. Developers who forked a repository can ask the owner of the original repository to merge the changes of the forked repository into the original one by submitting a *pull-request*. The owner of the original project can accept or deny the changes of the forked repository. In case of accepting a pull-request, the new changes are merged to the original repository. We consider both direct merges and pull-requests in this thesis.

2.2 Prediction Using Machine Learning

Providing algorithmic solutions is not possible for all types of problems. While some problems have sound and complete solutions (*e.g.* such as parsing code given a grammar and finding the shortest path between two nodes in a graph), proposing a deterministic algorithm may not be straight-forward for certain problems. For instance, although there is no known algorithm for detecting buggy classes so far, there are known bugs in the development history of different software repositories that we can use to predict the unknown bugs in the future using machine learning algorithms. The characteristics of each class are *features* (*e.g.* the number of lines of code and code complexity), and statuses of classes (buggy or not buggy) are *labels* of the features [59].

Supervised machine learning algorithms receive the data as input and as an output, they create a mathematical model to predict the label of unseen observations. While it is important to evaluate machine learning models using unseen data, these models often can generalize only for the unseen data that is *close* to the data they previously trained on. Despite the fact that the aim is training a model with generalization power, one usual problem in machine learning is that models *memorize* the training data, instead of *learning* the common patterns in it. This problem is so-called *over-fitting* [63]. Choosing the right values for internal parameters, *i.e.* hyper-parameters, regularization, and cross-validation, are common ways of avoiding over-fitting [65], [66].

Machine learning techniques can be categorized in many ways, here we describe some of the most common ones:

- *Unsupervised Learning:* When there is no label available for the data, we still can extract information from it at some level. This can include representation learning [12] to have a better input for a supervised model, data visualization [48], [50], [70], and data clustering to find natural partitions in the data [30], [35].

- *Supervised Learning:* This type of learning is employed when there is a label to predict. Classification [8] and regression [47] are the most well-known supervised problems. The former is when labels are discrete and finite, and the latter is when labels are real-valued numbers or vectors.

- *Semi-supervised Learning:* Labelled data can generally reveal more information than unlabelled one, but labelling the data is not cheap. Therefore, semi-supervised learning [20] uses both supervised and unsupervised data at the same.

- *Active Learning:* Since data labelling is an expensive task in some situation, it is important to start labelling from the observations (data points) that are most informative. Active learning aims to prioritize the data points and ask the agent (usually a human) to label the important ones first [26], [55].

- *Reinforcement Learning:* When there is no label available for the data, but the model receives feedback as a reward from the environment for each action that it takes. The goal of reinforcement learning is to learn a strategy to receive the highest amount of accumulative award [37].

Building a machine learning model contains two stages, *training* and *testing*. The training data is used to learn the parameters of the machine learning models and the testing data is used to report the results. Since the goal of machine learning is a generalization for unseen data, rather than memorizing the training data, the training and testing data should be exclusive.

In this thesis, we consider each merge scenario, with all involved commits, as an observation. We represent each merge scenario as a fixed-length real-

Table 2.1: The Confusion Matrix for Binary Classification

		Actual Classes	
		Target Class	***Not-Target Class***
Predicted Classes	***Target Class***	True Positive (TP)	False Positive (FP)
	Not-Target Class	False Negative (FN)	True Negative (TN)

valued feature vector. We then create our predictors using binary classification models to predict if conflicts can occur in each merge scenario.

2.2.1 Binary Classification Techniques

In a binary classification problem, the labels can hold binary values, 0 or 1, which means that the model should label a data point as either of these two classes. In our problem definition, 0 means there is no conflict in a merge scenario, and 1 means there is at least one merge conflict in a merge scenario. Therefore, we predict the existence of conflicts, rather than the number of conflicts in merge scenarios.

Similar to most of the machine learning techniques, a binary classification technique aims to minimize a cost function which tells the model how it needs to update the parameters to classify the given data better. For this purpose, there is a vast range of metrics defined to evaluate a binary classifier via the testing data. By comparing the actual labels of the training data and the prediction of a binary classifier, we can use the following metrics, as we also show in Table 2.1:

- *True Positive (TP):* The target class is labelled correctly.
- *False Positive (FP):* The non-target class is incorrectly labelled as the target class.
- T*rue Negative (TN):* The non-target class is labelled correctly.
- *False Negative (FN):* The target class is incorrectly labelled as the non-target class.

The most well-known binary classification metrics are defined using the above definitions, as follows:

$$accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{2.1}$$

$$recall = \frac{TP}{TP + FN} \tag{2.2}$$

$$precision = \frac{TP}{TP + FP} \tag{2.3}$$

$$f1 - score = 2 * \frac{precision * recall}{precision + recall} \tag{2.4}$$

2.2.2 Classification for Imbalanced Data

Imbalanced data means there are more data points in one class than the other one. Since conflicts do not occur frequently (based on the statistics in the following chapters), our data is imbalanced. This situation causes problems in two ways:

1. *Training:* Most of the machine learning techniques are not designed to handle imbalanced data in their original form. Therefore, using a model without any modification may result in having a bias towards the majority class, *i.e.* the class with more data points.

2. *Evaluation:* While accuracy (Equation 2.1) is a widely used criterion to evaluate a binary classifier, its output is not valid for imbalanced data. As an extreme example, if 99% of data points are in majority class and only 1% in the minority one, a hypothetical weak classifier that always labels the data as the majority class would achieve 99% accuracy. While this number seems to be impressive, it does not show the effectiveness of the classifier and it is misleading.

There are multiple ways to deal with imbalanced data in machine learning, including:

1. *Resampling the data:* The goal of this method is balancing the number of data points in two classes. The first way of doing this is *under-sampling* that keeps the data of the minority class and randomly select the equal size of the minority class form the majority class. This method eliminates parts of the available data of the majority class and therefore, should be used with caution. Another sampling strategy is *over-sampling* that keeps the data of the majority class and generates data for the minority class using repetition, bootstrapping, or Synthetic Minority Over-Sampling Technique (SMOTE) [21].

2. *Using the right evaluation criteria:* Since using accuracy is potentially misleading for evaluating a model using imbalanced data, other metrics such as f1-score are used as the accuracy's alternative. Furthermore, using cost-sensitive metrics helps the classifier to better consider the miss-classification of the minority class by magnifying the penalty of this class.

3. *Using ensemble learning:* Using multiple (and usually weaker) classifiers can decrease the bias towards the majority class [42]. Bagging [15] and boosting [33] are two common ways of creating such models.

In this work, we use ensemble learning models to achieve better classifier models. Furthermore, we use cost-sensitive metrics by employing the imbalanced rate to increase the effect of the miss-classification cost of conflicting merges which are the minority class. Finally, we report recall, precision, and f1-score metrics.

2.3 Classification Models

Here we briefly describe the classification techniques that we used in our work.

2.3.1 Decision Tree

This is a very simple, yet effective classification technique [44], [60], [61]. This classifier is also famous for its interpretability since its output is a set of rules.

Each non-leaf node in a decision tree is a condition on the value of a feature. The higher-level nodes select the features that can classify the data better. The entropy of the children of a node is used as a proxy of the effectiveness of that node. A feature that can classify the data into the nodes with the lowest entropy is considered as a better feature and therefore, decision trees are also useful to determine the importance of features. Increasing the number of levels increases the chance of over-fitting. Hence, there is usually a maximum number of levels as a hyper-parameter. Another important hyper-parameter in decision trees is the minimum number of data points in a node to expand.

2.3.2 Random Forest

This classification technique uses the ensemble learning approach to enhance performance. It trains multiple decision trees, each of them via a subset of features, and then assigns a label to an unseen data point by voting among all of the trees. Using the combination of weaker classifiers is shown to be a proper way of improving the classification result [46], [68]. In random forests, all of the data points and decision trees have the same weight in the voting process. Usually, the decision trees are the same in terms of hyper-parameters and their input data (which subset of features to use) is their only difference.

Chapter 3

Related Work

In this chapter, we provide an overview of the related work to our proposed method for predicting textual conflicts.

3.1 Merging Methods

We first provide a summary of existing merging techniques. For a more comprehensive classification, we refer the reader to Mens' survey on software merging [53].

Git is an *unstructured* merging tool that is language-independent and does not consider the structure of the code (or any underlying tracked file); when the same text in a file has been simultaneously edited, Git reports a *textual conflict*.

On the other hand, *structured merge* tools [18], [71], *e.g.*, FSTMerge [29], leverage information about the underlying code structure through analyzing the corresponding Abstract Syntax Tree (AST). Since differencing a complete AST is expensive, *semi-structured* merge tools, such as JDime [36], improve performance by producing a partial AST that expands only until the method level, with complete method bodies in the leaves. Structured merge is then used for the main nodes of the tree, while unstructured merge is used for the method bodies in the leaves.

In this thesis, we focus on textual conflicts as reported by Git, since these are the most common types of conflicts developers face in their typical workflow. We often use only the term *conflict* for brevity.

3.2 Empirical Studies on Software Merging

Previous studies compared the above merge techniques in practice in terms of speed, quality of resolutions, and the complexity of reported conflicts. For example, Cavalcanti et al. [19] focused on unstructured and semi-structured merge tools and found that using semi-structured merge significantly reduces the number of conflicts. The authors also found that the output of semi-structured merge is easier to understand and resolve. In follow-up work, Accioly et al. [6] investigated the structure of code changes that lead to conflicts with semi-structured tools. The study showed that in most of the conflicting merge scenarios, more than two developers are involved. Moreover, this study showed that code cloning can be a root cause of conflicts. While semi-structured merge is faster than structured merge and more precise than unstructured merge, it is still not used in the software industry due to the effort that is needed to support new programming languages. A recent large-scale empirical study by Ghiotto et al. [52] also investigated various characteristics of textual merge conflicts, such as their size and resolution types. The results suggest that since merge conflicts vary greatly in terms of their complexity and resolutions, having an automatic tool that can resolve all types of conflicts is likely not feasible.

One approach for reducing the resolution time is selecting the right developer to perform the merging based on their previous performance and changes [22]. Other work looked at specific types of changes that may affect merge conflicts. For example, Dig et al. [24] introduced a refactoring-aware merging technique that can resolve conflicts in the presence of refactorings. A recent study also shows that 22% of the analyzed Git conflicts involved refactoring operations in the conflicting code [49].

3.3 Speculative Merging

Given the cost of merge conflicts and integration problems, many research efforts have advocated earlier resolution of conflicts [17], [32], [64]. Previous

work has shown that lack of awareness of changes being done by other developers can cause conflicts [27], and since infrequent merging can decrease awareness, it increases the chance of conflicts.

To address that, *proactive merge-conflict detection* warns developers about possible conflicts before they attempt to merge, *i.e.*, before they try to push their changes or pull new changes. With proactive conflict detection, developers get warned early about conflicts so they can resolve them soon instead of waiting till later when they get large and complicated.

In the literature, proactive conflict detection is typically based on *speculative merging* [11], [16], [32], [40], where all combinations of available branches are pulled and merged in the background. While a single textual merge operation is computationally inexpensive, constantly pulling and merging a large number of branch combinations can quickly get prohibitively expensive. One opportunity we foresee for decreasing this cost is to avoid performing speculative merging on *safe merge scenarios* that are unlikely to have conflicts. To accomplish this, we can leverage machine learning techniques to design a classifier for predicting merge conflicts. The question we address in this thesis is whether such a classifier works well in practice.

3.4 Proactive Conflict Detection

There are several approaches to increase the awareness of developers by detecting conflicts early. Awareness of changes other team members may be making is a key factor in team productivity and reduces the number of conflicts [27]. Syde [34] is a tool for increasing awareness through sharing the code changes present in other developers' workspaces. Similarly, Palantir [64] visually illustrates code changes and helps developers avoid conflicts by making them aware of changes in private workspaces. Crystal [17] is a visual tool that uses speculative analysis to help developers detect, manage, and prevent various types of conflicts. Cassandra [40] is another tool to minimize conflicts by optimizing task scheduling, to minimize simultaneous edits to the same files. MergeHelper [56] helps developers find the root cause of merge conflicts

by providing them with the historic edit operations that affected a given class member.

Guimarães et al. [32] propose to continuously merge, compile, and test committed and uncommitted changes to detect conflicts as early as possible. However, such an approach is likely expensive given a large number of combinations of branches and developer changes in large projects.

3.5 Merge Conflict Prediction

To the best of our knowledge, there is no research work to study the possibility of predicting the existence of conflicts in unseen merge conflicts using machine learning. However, two main studies try to investigate the relationship between the features of merge scenarios (*e.g.* code and Git history features) and conflicts.

Accioly et al. [7] investigate whether the occurrence of events such as *edits to the same method* and *edits to directly dependent methods* can be used to predict conflicts. However, they do not build a prediction model. Instead, they count the number of times each of the above features exists when a conflict occurs versus when the merge is successful. Based on such counts, their results show a precision of 57.99% and a recall of 82.67%.

Leßenich et al. [45] investigate the correlation between various code and Git features and the likelihood of conflicts. To create a list of features they investigated, they first surveyed 41 developers. The developers mentioned seven features that can potentially cause conflicts. However, after analyzing 21,488 merge scenarios in 163 Java repositories, the authors could not find a correlation between these features and the likelihood of conflicts. We speculate that one reason for not capturing such relationships is using stepwise-regression which may not be an effective model for imbalanced and non-linear data, such as that collected from merge scenarios.

In this thesis, we investigate merge conflict prediction by creating a list of nine feature sets that can potentially impact conflicts. Our list is based on previous work in the areas of software merging and code review [27], [28], [43],

[45], [64]. Our work is different from all the above in that we use statistical machine learning to create a classifier, for each repository, that can predict conflicts in unseen merge scenarios.

3.6 Data Acquisition in Software Merging

Researchers study merge scenarios to design better development tools or to introduce better practices to reduce the number of conflicts. Empirical studies on software merging can also shed light on the characteristics of current software development practices and ways of improving them. There is a number of previous work that analyzes the performance and functionality of different merging techniques [6], [19], predicts merge conflicts [7], [45], detects conflicts early [17], [28], [32], analyzes the merging status of pull-requests [43], [72], or studies the code review process associated with pull-requests [10], [23], [69].

Over the last couple of decades, several tools were proposed for mining software repositories, especially for Git and GitHub repositories. However, to the best of our knowledge, there is no off-the-shelf tool that focuses on software merging. Boa [25] is a tool, with an accompanying language, for running large-scale queries on data from GitHub and SourceForge. However, it queries snapshots of these websites, rather than real-time data.

GHTorrent [31] is an offline mirror of GitHub that allows users to either download the data as a SQL or MongoDB database or run their queries online. PyDriller [67] is a recent tool for analyzing Git history to extract data such as commits, developers, source code, etc. GitMiner [1] is an open-source tool that stores data extracted from Git and GitHub in a database.

Although both PyDriller and GitMiner can be used to detect merge commits by selecting commits with more than one parent, neither provides any additional option to analyze merge scenarios or merge conflicts. GrimoireLab [2] is an industrial-level tool that is capable of gathering data from version control systems, issue trackers, mailing lists, wikis, but it does not contain any tooling for analyzing merge scenarios.

Some papers that study software merging (e.g., [6]) release the tool used

for mining the merge data. However, the tool is specific to the study and cannot be directly employed for general software merging research due to lack of scalability and covering only a limited number of merge-scenario features extracted for the specific study.

Since there is not a scalable and well-tested tool to extract the data of software merging, we develop our tool, MergAnser, which is capable of extracting the data of merges and conflicts as a normalized SQL database. In the next chapter, we introduce this tool and its features.

Chapter 4

Data Collection with MergAnser

In this chapter, we introduce our data acquisition tool, MergAnser [3], which we develop and release as an open-source tool under an `MIT` license. We also introduce the data we collected using MergAnser for our study, the criteria for choosing repositories, and the relevant statistics of our data.

4.1 MergAnser

MergAnser is a scalable tool that we develop and maintain to help us gather the data of merges on GitHub as a normalized SQL database. This tool not only allows us to collect the data we need for our textual conflict prediction models but also can be employed by other researchers who need clean and normalized data to study merge scenarios.

Given a list of repositories, MergAnser extracts the information of each repository such as their popularity metrics (*i.e.* the number of stars, forks, watches) and their description using the standard GitHub APIs. Then, to gather the necessary data of merge scenarios, MergAnser first clones the repositories locally and then detects merge scenarios using Git commands. We consider any commit with two parents as a merge scenario, which includes direct merging and pull-requests. Our detection strategy means we miss rebased merge scenarios since rebasing creates a linear history. However, there is currently no accurate technique for detecting merge scenarios that have been rebased. In the next step, MergAnser replays each merge commit to detect the conflicting files and regions. The reason we replay each merge scenario is

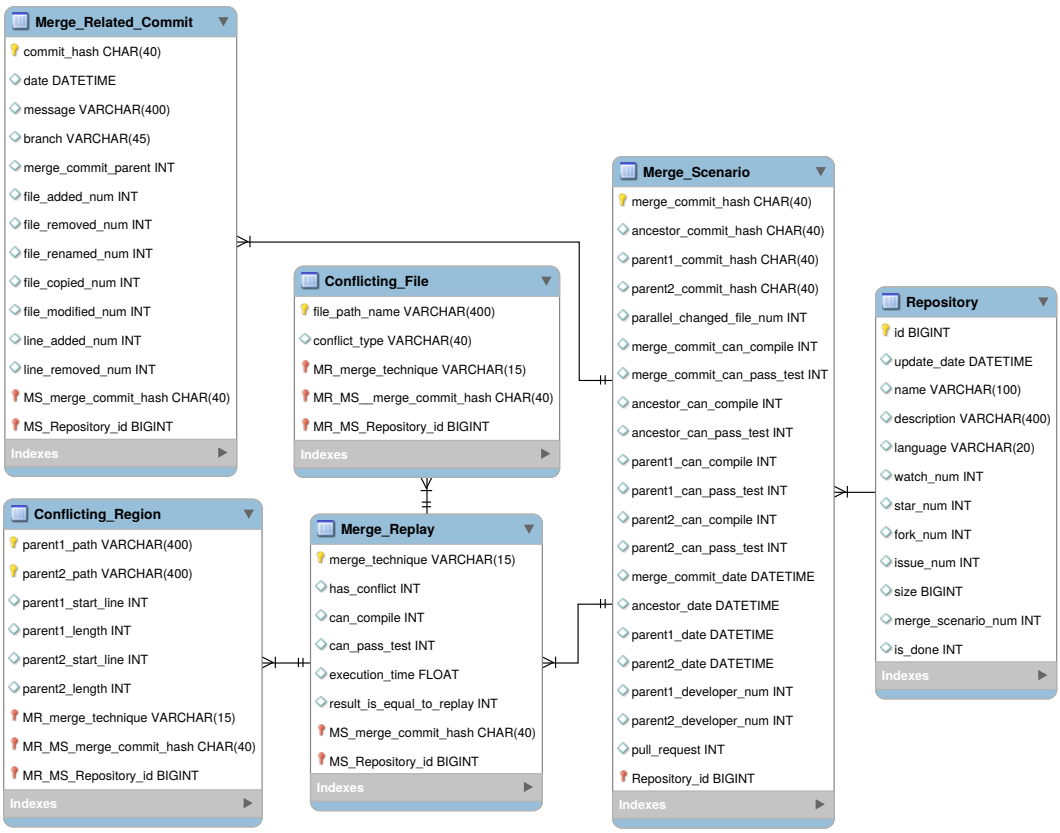

Figure 4.1: The MergAnser database schema

that the information of merge conflicts is not stored in the Git history. Finally, MergAnser stores all the extracted information in a SQL database, according to the schema in Figure 4.1.

In Table 4.1, we summarize the description of each table in our database schema. We use GitHub APIs and Git commands to extract the data of the `Repository` table and only use Git commands to extract the data of other tables. The exact details of the tools and commands we use are available in MergAnser's online documentation [4].

MergAnser can analyze multiple repositories in parallel, using as many CPU cores as the user specifies. We designed the data schema to be easily extensible, such that extracting any new features in the future is easy.

Table 4.1: The list of tables in the MergAnser database schema

No.	Table Name	Description
1	`Repository`	Data of analyzed repositories, *e.g.* the analysis date, name, description, and programming language
2	`Merge_Scenario`	Data of merge scenarios, *e.g.* the SHA-1 and the date of the ancestor, parents, and the merge commit
3	`Merge_Related_Commit`	Data of all commits that are involved in the merge scenario, *e.g.* the SHA-1, date, commit message, and branch name
4	`Merge_Replay`	MergAnser replays merge scenarios to detect conflicts and stores their characteristics, such as whether the merge scenario has any conflict
5	`Conflicting_File`	Data of all files that have conflicts, including their relative path and the type of conflict reported by Git (*e.g.*, content conflict vs. delete/modify)
6	`Conflicting_Region`	Data of conflicting regions, including the paths of the two parents' files and the location of the conflict region, represented as the start line and length of the region

4.1.1 MergAnser Usage Examples

To provide more insights into the potential usages of our MergAnser, we provide three sample queries here to extract relevant information from the data extracted by MergAnser.

Simultaneously Changed Files To extract the number of files that are edited in two branches in parallel, the user can run the following query:

```
SELECT merge_commit_hash, parallel_changed_file_num
FROM Merge_Data.Merge_Scenario
```

The number of commits The following query extracts the number of involved commits in merge scenarios in Java:

```
SELECT merge_scenario.merge_commit_hash, COUNT(commits.commit_hash
    )
FROM Merge_Data.Repository as repository
JOIN Merge_Data.Merge_Scenario as merge_scenario
    ON repository.id = merge_scenario.Repository_id
JOIN Merge_Data.Merge_Related_Commit as commits
    ON merge_scenario.merge_commit_hash  = commits.
    MS_merge_commit_hash
WHERE repository.language = 'java'
GROUP BY merge_scenario.merge_commit_hash
```

The number of added/removed lines To extract the difference between the number of lines that are added and deleted in two branches:

```
SELECT merge_scenario.merge_commit_hash,
        SUM(commits.line_added_num *   IF(commits.
    merge_commit_parent=2,1,-1)) AS 'line_added',
        SUM(commits.line_removed_num * IF(commits.
    merge_commit_parent=2,1,-1)) AS 'line_removed'
FROM Merge_Data.Merge_Scenario as merge_scenario
JOIN Merge_Data.Merge_Related_Commit as commits
    ON merge_scenario.merge_commit_hash  = commits.
    MS_merge_commit_hash
GROUP BY merge_scenario.merge_commit_hash
```

4.2 The Dataset of Merge Scenarios

We describe the systematic procedure we follow to gather the data of this study in this section. This includes the selection of target repositories to analyze and extracting merge scenarios.

4.2.1 Selecting the Target Repositories

The first step to collect the merge scenarios for our study is to choose the target repositories to be analyzed. We focus on open-source repositories in this thesis and therefore, need to ensure that the selected repositories are of high quality and reflect real-world development practices. As a proxy for quality, we look for well-engineered repositories (*i.e.*, real-world engineered software projects [54]) that are also popular. Specifically, we use the following criteria:

- ***Popularity:*** Intuitively, more active and useful repositories attract more attention, reflected in the number of stars, issues, and forks. Similar to the previous studies[6], [7], we use the number of stars as a filtering criterion.

- ***Quality:*** Even though the number of stars represents some measure of quality, not all popular repositories are suitable for our study. For instance, there are a number of repositories that only consist of code examples and interview questions that are highly starred but are not suitable for studying merge conflicts since they do not represent a collaborative effort to build a software system. Hence, we apply further quality measures for our repository selection. We use reaper [54] to detect well-engineered software repositories and avoid analyzing personal or toy repositories. Reaper uses various repository characteristics such as community support, continuous integration, documentation, history, issues, license, and unit testing to classify well-engineered software repositories using a random forest classier. We use the reaper's released dataset [62] (downloaded on September 15, 2018) and select all repositories in that list that have been classified as well-engineered repositories.

- ***Programming Language:*** We choose all seven programming languages that the reaper dataset supports: C, C#, C++, Java, PHP, Python, and Ruby.

Considering the three criteria mentioned above, we sort the well-engineered repositories in each programming language separately based on the number of stars. We then select the top 100 repositories from each language, for a total of 700 repositories as the initial list. However, we need to eliminate a subset of repositories for the following reasons:

1. For practical limitations with respect to computational resources for replaying thousands of merge scenarios from that many repositories, we need to eliminate some of the repositories in the initial list.

2. Machine learning algorithms need adequate data for training and testing. Since we train a binary classifier for each repository, it is important to have enough merge scenarios for training and testing in the target repositories.

Considering the above-mentioned reasons, we eliminate the repositories in the following categories according to Table 4.2:

1. To avoid analyzing the same merge scenario multiple times, we only analyze the main repositories and eliminate the forked versions.

2. We focus on active repositories and therefore eliminate any moved or archived repositories from that initial list.

3. To have enough data for training and testing a binary classifier, we eliminate the repositories that have less than 200 merge scenarios in their history.

4. There should be conflicting merge scenarios in both training and testing data and hence, we eliminate the repositories that do not satisfy this condition.

5. The test data should have enough data to be able to report a reliable classification report. For this reason, we only consider the repositories that have more than 10 conflicting merge scenarios in the testing data.

6. To make the evaluation practical, we only consider repositories whose size is less than 1 GB and focus on the latest $5,000$ merge scenarios in each repository.

Therefore, as we show in Table 4.2, only 105 (out of 700 repositories in our initial list) repositories are suitable to use for training and testing our conflict predictors. Table 4.3 provides some descriptive statistics of these repositories. Moreover, since we use the number of stars as an indicator of the popularity of repositories, the distribution of the number of stars that each repository has is illustrated across different programming languages in Figure 4.2.

Table 4.2: The status of repositories we used in our study (the minus sign (−) shows the deductions)

#	Item	No. Repositories
1	The initial list of repositories	700
2	Forked repositories	4 (−)
3	Moved/archived repositories	91(−)
4	Repositories with less than 200 merge scenarios	290(−)
5	Repositories without conflicting merges in the training/testing data	30(−)
6	Repositories with less than 10 conflicting merge scenarios in the testing data	144(−)
7	Executions terminated with errors	36(−)
8	**The final list of repositories for experiments**	105

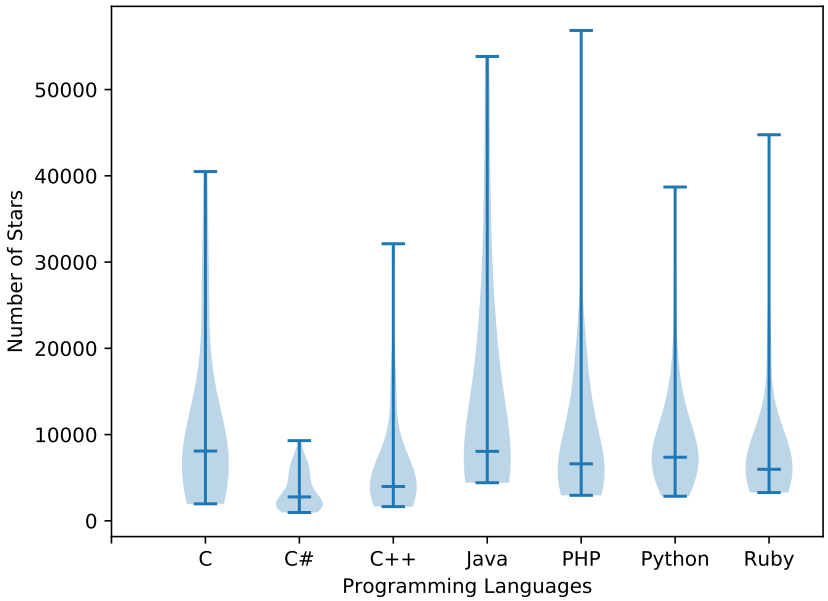

Figure 4.2: The distribution of the number of stars across different programming languages

Table 4.3: Descriptive Statistics of the Selected Repositories

#	Statistics	Min	Average	Max
1	No. Stars	967	8,975.91	86,852
2	No. Watches	32	497.68	4,636
3	No. Forks	209	2,300.61	18,950
4	No. Issues	0	339.80	7,368
5	Size (MB)	0.93	91.24	991.35

4.2.2 Extracting Merge Scenarios

After choosing the target repositories, we extract their latest 5,000 merge scenarios using MergAnser. We collect 147,967 merge scenarios in total. Figure 4.3 illustrates the distribution of merge scenarios for repositories in dif-

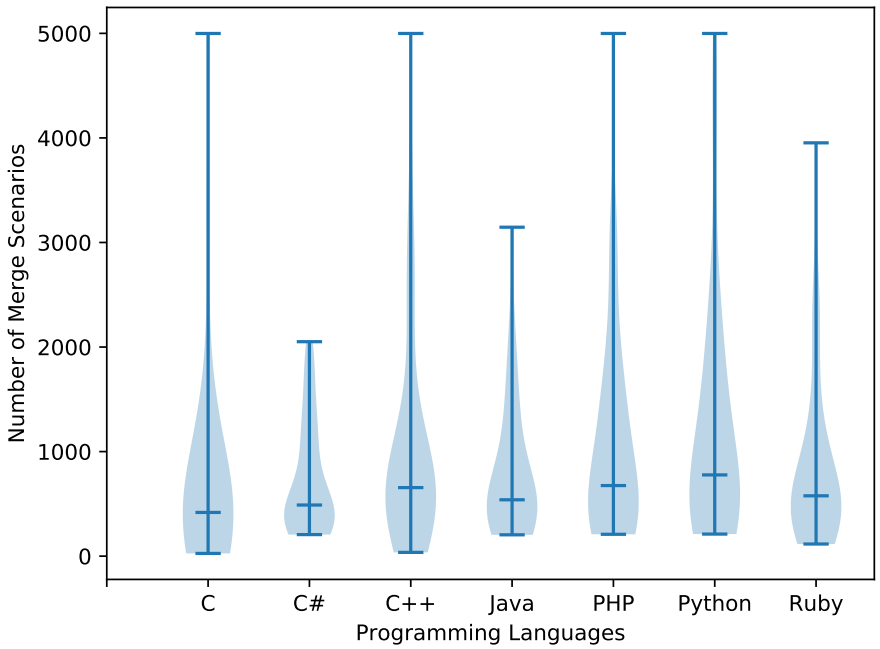

Figure 4.3: The distribution of the number of merge scenarios across different programming languages

ferent programming languages. The median of the number of merge scenarios over different programming languages is close to each other, around 500. In Figure 4.4, we illustrate the distribution of the number of conflicting merge scenarios in the same way. Similarly, Figure 4.5 shows the conflict rate, *i.e.* the portion of merges that have at least one conflict. The median of this rate for all programming languages is less than 10%.

Merge scenarios can be identified from a repository's Git history. However, unfortunately, not all information about a merge scenario (*e.g.*, whether there was a conflict or not) is available in Git's data. Therefore, to identify conflicting merge scenarios and determine whether the merge resulted in a conflict, we use a *replaying* method where we re-perform the merge at that point of history and record the outcome. For each merge scenario, we detected the parent commits and replay the merge locally using `git merge` command. We then check the Git standard output to see if there is any textual conflict in the merge scenario and store the relevant information in the database.

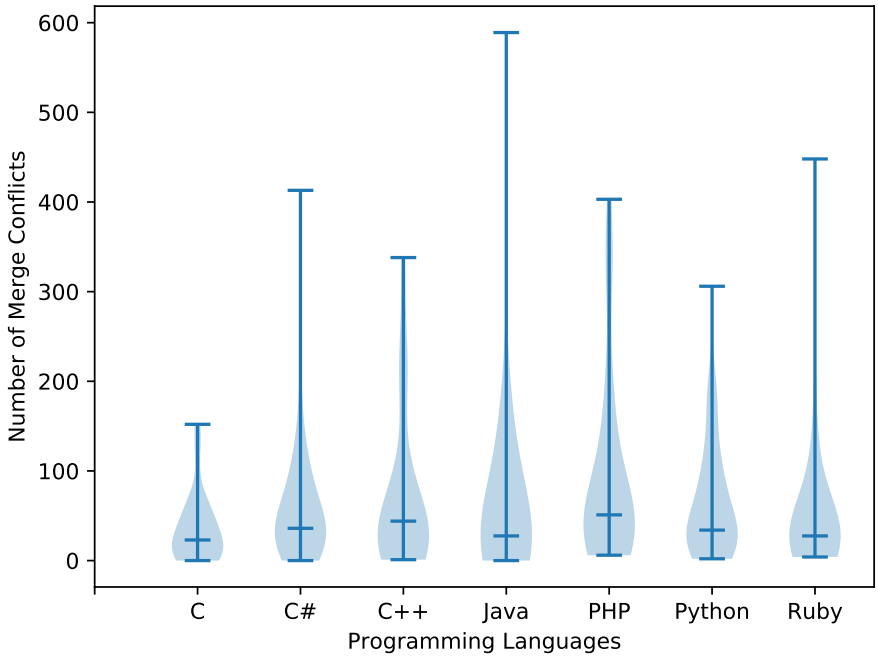

Figure 4.4: The distribution of the number of conflicting merge scenarios across different programming languages

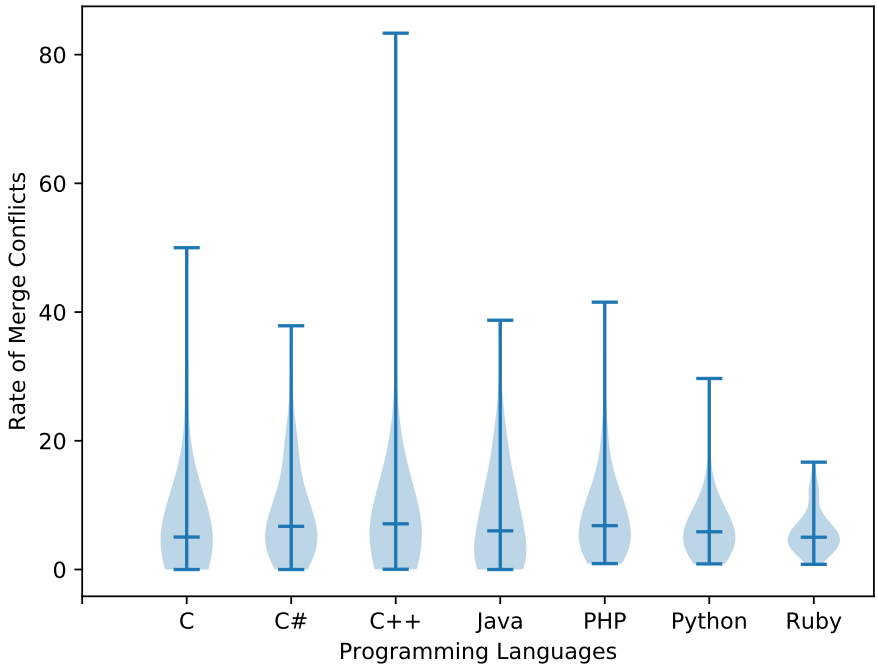

Figure 4.5: The distribution of conflict rates across different programming languages

Chapter 5

Feature Extraction

We introduce the features we extracted from our data in this chapter. We selected the list of the features to extract from a merge scenario based on analyzing the kind of information previous studies on software merging typically need [27], [28], [43], [45].

To train a classifier, we need to extract potentially predictive features from merge scenarios. Our goal is to use features whose extraction is computationally inexpensive such that the prediction can be used in practice. We identify these features by surveying the literature on merge conflicts and related areas, such as code evolution or software maintenance.

In Table 5.1, we categorize the identified features into 9 *feature sets*, along with the intuition behind them, as well as any relevant related work that previously used this feature set or a variation of it. For example, feature set #4 is inspired by previous merging and code review studies [28], [43] and indicates changes in files. We include this feature set since more code changes may increase the chance of conflict. We do not rely on language-specific features; all of our feature sets are language-agnostic.

The feature sets shown in Table 5.1 are on different granularity levels. Feature set #1, *No. of simultaneously changed files*, is a *merge-level* feature set which means that this feature set is extracted once for a given merge scenario. All other feature sets are in *branch-level* which means that they are extracted from each branch separately. Each feature set should have a single value or vector for each merge scenario. This means that we need

Table 5.1: Feature Sets Used For Training the Merge Conflict Predictor

No.	Feature Set	Intuition for Including this Feature Set
1	No. of simultaneously changed files in two branches	The increase in simultaneously changed files increases the chance of conflict. If the value of this feature is zero, no conflict can occur [45], [64]
2	No. of commits between the ancestor and the last commit in a branch	Having more commits means more changes in a branch, which may increase conflicts [28], [43], [45]
3	Commit density: No. of commits in the last week of development of a branch	Lots of recent activities may increase the chance of conflicting changes [45]
4	No. added, deleted, renamed, modified, and copied files in a branch	More code changes may increase the chance of conflict [28], [43], [45]
5	No. added and deleted lines in a branch	More code changes may increase the chance of conflict [28], [45]
6	No. of active developers in a branch	Having more developers increases the chance of getting inconsistent changes [27], [28], [43]
7	The frequency of predefined keywords in the commit messages in a branch. We use 12 key-words: fix, bug, feature, improve, document, refactor, update, add, remove, use, delete, and change	These keywords can provide a high-level overview of the types of code changes and their purpose. Certain types of changes may be more prone to conflicts[28]
8	The minimum, maximum, average, and median length of commit messages in the branch	The length of a commit message can be an indicator of its quality [28]
9	Duration of the development of the branch in hours	The longer a branch exists for, the more likely it is for inconsistent changes to happen in one of the other branches [27]

to combine the two values of branch-level feature sets. Since the choice of the combination operator may impact the performance of the classifier, we empirically determine the best combination operator to use, as we describe in the following chapters.

Table 5.2 shows the dimension of each feature set (*i.e.*, the number of

Table 5.2: The Dimension (D) of Each Feature Set

No.	**Feature Set**	**D**
1	No. of simultaneously changed files in two branches	1
2	No. of commits between the ancestor and the last commit in a branch	1
3	Commit density: No. of commits in the last week of development of a branch	1
4	No. added, deleted, renamed, modified, and copied files in a branch	5
5	No. added and deleted lines in a branch	2
6	No. of active developers in a branch	1
7	The frequency of predefined keywords in the commit messages in a branch. We use 12 key-words: fix, bug, feature, improve, document, refactor, update, add, remove, use, delete, and change	12
8	The minimum, maximum, average, and median length of commit messages in the branch	4
9	Duration of the development of the branch in hours	1
Total features		**28**

individual values, each corresponding to a feature, used as input to the model) for the prediction task. The dimension of some of these feature sets is one, which means that they are just a scalar value. Some other feature sets have a dimension greater than one in order to represent all the needed information; such feature sets would be represented as a vector. For instance, feature set #4 needs five values to represent the number of added, deleted, modified, copied, and renamed files. Feature sets #4, #5, #7, and #8 are vectors with size 5, 2, 12, and 4, respectively, and the other feature sets are scalars. In the end, each merge scenario is represented by a total of 28 features.

Chapter 6

RQ1: Which characteristics of merge scenarios have more impact on conflicts?

In RQ1, we are interested in identifying which feature sets are more important for predicting conflicts. In this chapter, we first describe the analysis methods we use, given the features extracted in Chapter 5 and then present the results.

6.1 Methodology

To answer RQ1, we analyze the 9 feature sets in Table 5.1 to see which of them are more important. We analyze importance in two ways:

1. ***Correlation Analysis:*** We calculate Spearman's rank-order correlation[41] which is a non-parametric measure between the feature sets and the existence of conflicts. This is the same correlation method used in previous work to determine the effectiveness of various features for predicting conflicts [45].

2. ***Supervised Analysis:*** We use decision trees to analyze the importance of each feature set since the results of decision trees are easier to interpret than other classifiers. A decision tree aims to find a single feature set in each level based on which it can classify the data in the most optimized way. For feature sets that have more than one feature, we calculate the average of the importance of their individual features.

Table 6.1: Spearman's Rank-Order correlation coefficients (CC) and their corresponding p-Values (p). CC with $p < 0.05$ are highlighted.

Programming Languages		**Feature Sets**								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
C	***CC***	0.45	0.19	0.0	-0.02	-0.02	0.03	0.0	0.12	-0.01
	p	0.0	0.0	0.06	0.05	0.16	0.14	0.14	0.01	0.43
C#	***CC***	0.54	0.26	-0.07	-0.05	-0.04	0.0	-0.03	0.12	0.0
	p	0.0	0.0	0.02	0.05	0.12	0.11	0.1	0.0	0.27
C++	***CC***	0.49	0.21	-0.03	-0.03	-0.03	0.0	-0.01	0.11	-0.02
	p	0.0	0.0	0.09	0.11	0.04	0.07	0.04	0.01	0.24
Java	***CC***	0.55	0.26	-0.05	-0.04	-0.03	0.01	0.1	0.1	-0.02
	p	0.0	0.0	0.02	0.03	0.01	0.26	0.7	0.0	0.35
PHP	***CC***	0.55	0.29	-0.08	-0.07	-0.09	-0.01	-0.07	0.14	-0.01
	p	0.0	0.0	0.0	0.01	0.0	0.09	0.0	0.0	0.24
Python	***CC***	0.5	0.28	-0.04	-0.04	-0.04	-0.01	-0.03	0.11	-0.01
	p	0.0	0.0	0.01	0.04	0.05	0.14	0.03	0.0	0.45
Ruby	***CC***	0.52	0.28	-0.07	-0.05	-0.03	0.3	-0.3	0.14	0.0
	p	0.0	0.0	0.03	0.05	0.12	0.2	0.07	0.0	0.3

6.2 Results

6.2.1 Correlation Analysis

We first analyze the Spearman's Rank-Order correlation between the feature sets and merge conflicts, as shown in Table 6.1. We calculate the correlation and the corresponding p-value for each feature set separately.

Following previous work [45] and statistics guidelines [9], we consider correlation coefficients ≥ 0.6 as strong , $0.4 - 0.59$ as medium, and $0.2 - 0.39$ as weak. We only consider statistically significant correlations whose p-value ≤ 0.05 and highlight them in the Table 6.1.

The p-values of feature set #9, for all languages, are greater than 0.05 showing that there is no significant correlation between the conflicts and these feature sets. The p-values of feature set #1, #2, and #8 for all programming languages are less than ≤ 0.05. However, none of the feature sets show a strong correlation and only Feature Set #1 has a medium correlation for all programming languages. There are also weak statistically significant correlations between feature set #2 and conflicts for all programming languages,

except C. In Ruby repositories, there is a weak correlation for the feature set #7 but it is not statistically significant. For the rest of the cases, there are no significant correlations.

While we do not use the same exact features from the previous work by Leßenich et al.[45], we can confirm their findings in terms of lacking correlation between Git features of merge scenarios and conflicts. This gives us confidence that the lack of correlations we find for most features is correct. However, we argue that this lack of correlation does not necessarily mean merge conflicts are not predictable, as we show later in the results of RQ2.

Although we report the correlation and importance of feature sets for different programming languages separately, it is important to note that we do not expect to see significant differences between the feature sets in different programming languages since our feature sets are language-agnostic. Our results in Table 6.1 confirm that.

6.2.2 Supervised Analysis

As a different way of measuring feature importance, we use decision trees to determine the importance of our feature set for predicting conflicts in each programming language. The results are shown in Table 6.2, where the feature importance is a value between 0 and 1. Again, we find that the number of simultaneously changed files (Feature Set #1) is the most important feature by far. The high impact of the number of simultaneously changed files can be intuitively explained since more in-parallel changes increases the likelihood of conflicts, and the chance of conflict is zero if there are no simultaneously changed files. The number of commits (Feature Set #2), active developers (Feature Set #6), and the frequency of keywords in commit messages (Feature Set #7) also have non-zero values for some programming languages. However, apart from Feature Set #1, all features have very low importance for the classifier. Similar to the correlation-based analysis, we find that the importance of feature sets is relatively similar for all programming languages.

Our results suggest that commit message information (feature sets #7 and #8) is not important for detecting conflicts. Since commit messages contain

Table 6.2: Feature Importance Based on a Decision Tree Classifier

Programming Languages	**Feature Sets**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
C	0.87	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0
C#	0.90	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
C++	0.91	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0
Java	0.88	0.0	0.0	0.0	0.0	0.0	0.04	0.0	0.0
PHP	0.90	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Python	0.94	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Ruby	0.94	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

information about the evolution of a repository (*e.g.*, indications of types of code changes), we intuitively thought that they may have an impact on conflicts. However, the feature sets related to commit messages we currently extract are intentionally lightweight to keep execution time low. It is, therefore, hard to conclude if commit messages are indeed altogether useless in this context or if different types of feature sets (*e.g.*, taking word embedding of the commit messages into account) may lead to more meaningful relationships. This finding is important since unlike the other numerical features, analyzing the commit messages is computationally expensive due to text processing.

The zero feature importance in Table 6.2, does not necessarily mean that there is no connection between the features with zero feature importance and conflicts. First, some of the values are not absolutely zero and we report them as zero because of rounding the numbers up to two digits after the decimal point. Second, decision trees are simple classifiers and might not be able to discover sophisticated patters in the data.

The number of simultaneously changed files is the most important feature for predicting merge conflicts. The number of commits in each branch shows a weak correlation, but a much lower importance level by the decision tree. The remaining feature sets show low correlation coefficients and importance.

Chapter 7

RQ2: Are merge conflicts predictable using only Git features?

In this chapter, our goal is to determine if merge conflicts can be predicted using our selected feature sets. We first describe the classifiers we use and then present the results.

7.1 Methodology

We use the features we extracted in Chapter 5 for training our conflict predictors, which are binary classifiers. We train and test binary classifiers for each repository separately and report the aggregated results for repositories in each programming language. For each repository, we sort its merge scenarios chronologically and use the first 70% of data for training and 30% for testing the predictors. It means that we use the older merge scenarios (the first 70%) to predict the newer unseen ones (the remaining 30%) and there is no data or feature leak through time in our predictors. We compare the performance of the decision tree and random forest classifiers we built using all 9 collected feature sets. Additionally, we also create two baselines to compare our results against. The following summarizes the four classifiers we compare.

1. ***Decision Tree:*** A cost-sensitive decision tree classifier is one of the simplest options for a binary classification task that is also robust to imbalanced data. We train a decision tree with all of our 9 feature sets.

Note that this is the same classifier used to determine importance in RQ1.

2. ***Random Forest:*** To investigate if a more sophisticated classifier can make more use of the available features, we use cost-sensitive random forests. In contrary to decision trees, random forests use random numbers in the classification process and therefore, changing the value of the random seed might have impact on the results. Thus, for each repository, we train and test the random forest classifier 10 times with 10 different random seeds that are randomly selected to report more reliable results. When reporting results, we calculate a single number for each repository, calculated as the average of the 10 runs.

3. ***Baseline #1:*** The first baseline we compare to is a "dummy" classifier that randomly labels the data. If the data was balanced, the f1-score of both classes would be around 0.5, *i.e.* random guess. However, since our data is not balanced, the f1-score of this classifier would be the same as the imbalance rate in the data for the conflicting class, *i.e.*, 0.10, and would be 0.90 for the not conflicting class. We expect that any other predictor should be better than this basic baseline to be useful in practice.

4. ***Baseline #2:*** Recall that in the results of RQ1 (Chapter 6), we found that feature set #1, which is the number of simultaneously changed files in two branches, is the most important feature for the decision tree classifier. Therefore, as our second baseline, we use a decision tree classifier that uses *only* feature set #1 from Table 5.1. The goal of this baseline is to have a low-cost classifier that relies only on the most important feature. That way, we can determine if having the other features improves things, or is simply an added cost with no benefit.

7.1.1 Hyper-parameters

The main hyper-parameters of decision trees and random forest classifiers are (1) the minimum samples in leaves, (2) the minimum sample split, (3) the maximum depth, and (4) the total number of estimators (just for random forest). Determining the proper values of hyper-parameters helps us to reduce the chance of over-fitting in our models. Due to the importance of these hyper-parameters, we use random search to find the right values of them. We use the range of 2 to 35 for the minimum samples in leaves and minimum sample split, 1 to 8 for the maximum depth, and 1 to 10 as the choices for the number of estimators.

7.1.2 Combination operators

Recall from Chapter 5 that since some of our feature sets are extracted for *each* branch separately, we need to use a combination operator to combine them into a single value for the whole merge scenario. To find the suitable combination operator to use, we train our predictors based on each of seven common combination operators: *Minimum*, *Maximum*, *Average*, *Median*, *Norm*-1 *of Difference*, *Norm*-2 *of Difference*, and *Concatenation* operators. We then use random search to determine the best combination operator. We find that Norm-1 is the best combination operator for all seven programming languages.

We use the scikit-learn [5] Python library to implement our predictors. Unless we mention, the pre-defined settings are used for tuning the classifiers. We report the median of our performance measures for both the conflicting and not conflicting (safe) classes in the following section.

7.2 Results

Table 7.1 shows our results for RQ2. Note that we do not show the results of Baseline #1 since it can be calculated based on the bias in the data and serves as a minimum threshold that any useful predictor needs to achieve. The numbers of each cell in Table 7.1 are the median of values for repositories in different programming languages. While we report the numbers for both

Table 7.1: Merge conflict prediction result comparing Baseline #2 (B2), decision trees (DT), random forests (RF) in terms of precision (P), recall (R), and f1-score (F1) - Highest f1-scores in each case are highlighted

Programming Languages	Models	Not Conflicting			Conflicting		
		P	R	F1	P	R	F1
C	***B2***	1.0	0.83	0.91	0.3	1.0	0.45
	DT	0.99	0.93	0.95	0.35	0.86	0.45
	RF	0.99	0.95	0.95	0.42	0.72	0.48
C#	***B2***	1.0	0.89	0.94	0.38	1.0	0.55
	DT	0.99	0.92	0.94	0.60	0.88	0.66
	RF	0.97	0.93	0.94	0.57	0.72	0.60
C++	***B2***	1.0	0.84	0.91	0.41	1.0	0.57
	DT	0.99	0.90	0.94	0.50	0.92	0.61
	RF	0.98	0.90	0.94	0.52	0.82	0.60
Java	***B2***	1.0	0.82	0.9	0.49	1.0	0.66
	DT	0.97	0.90	0.94	0.61	0.85	0.71
	RF	0.95	0.90	0.93	0.61	0.81	0.67
PHP	***B2***	1.0	0.85	0.92	0.44	1.0	0.61
	DT	0.99	0.90	0.94	0.50	0.90	0.63
	RF	0.98	0.92	0.94	0.51	0.78	0.59
Python	***B2***	1.0	0.81	0.89	0.29	1.0	0.45
	DT	0.99	0.89	0.93	0.42	0.85	0.53
	RF	0.98	0.93	0.95	0.46	0.76	0.52
Ruby	***B2***	1.0	0.86	0.93	0.34	1.0	0.51
	DT	0.99	0.93	0.95	0.43	0.79	0.53
	RF	0.97	0.95	0.96	0.44	0.69	0.52

conflicting and not conflicting classes, it is the latter that has the potential usage in filtering-out the *safe* repositories in speculative merging and therefore, it is important in our study. We interpret the results in the following.

7.2.1 Comparing Decision Trees and Baseline #2

We first compare Baseline #2, which is a simple decision tree classifier that uses the most important feature set determined in Chapter 6, to the decision tree classifier that uses all feature sets. Table 7.1 shows that the decision tree classifier that uses all features achieves a higher (or equal) f1-score for both

classes when compared to Baseline #2. This suggests that despite feature set #1 being the most important feature, adding the other features to the classifier does improve the results.

Additionally, both the decision tree classifier and Baseline #2 outperform Baseline #1 (the "dummy" classifier) in the not conflicting class. This shows that there is gained value in designing a "real" classifier.

7.2.2 Comparing Decision Trees and Random Forests

Given that the decision trees with all features outperform Baseline #2, we now compare the decision trees to random forests to determine if a more sophisticated classifier can achieve better results. The results in Table 7.1 show that decision tree models achieve the highest f1-score for both safe and conflicting merges in most cases. This shows that using a more advanced ensemble machine learning classifier did not help in achieving better results. It is however important to note that the margin of difference between the f1-score of the decision trees and random forests is small for most languages and this difference is mainly observed in the conflicting class. Another observation is that all the classifiers seem to perform consistently across the different programming languages.

7.2.3 The Results for the Conflicting class

We now focus on decision trees and discuss the results for the conflicting class in more detail. Table 7.1 shows that recall of the conflicting class ranges from 0.79 to 0.92 for the different programming languages. This means that the predictor can correctly identify most of the conflicting merge scenarios. The table shows that the precision of the conflicting class is in a lower range, varying from 0.35 to 0.61. On the one hand, having a high recall means that following the prediction of our proposed models, developers can perform early merging to prevent merge conflicts or at least prevent them from becoming complicated. On the other hand, a low precision means that our models suggest developers perform unnecessary merges. Considering the fact that early merging is not a harmful operation, while resolving conflicts is time-consuming and error-prone,

the extra merging does not have a notable negative impact on the development process. Overall, the f1-score of conflicting class ranges from 0.45 to 0.71 across the seven languages.

7.2.4 The Results for the Not Conflicting class

In terms of not conflicting (safe) merge scenarios, Table 7.1 shows that decision trees' recall for the not conflicting class is between 0.89 to 0.93. This is a high recall rate and means that the predictor can correctly identify most of the automatically mergeable merge scenarios (*i.e.*, those that will not result in conflicts). The precision of this class is between 0.97 to 0.99 for different programming languages, meaning that almost all of the merge scenarios that are predicted as safe are actually safe. Overall, the f1-score of the not conflicting class ranges between 0.93 to 0.95 for the different programming languages. The results of our models for the not conflicting class suggest that we can potentially use the output of our models to filter-out the safe merge scenarios in the speculative merging process.

We, finally, note that the average time for predicting the status of a given merge scenario, including the feature extraction process, is 0.1 ± 0.02 seconds. This makes our predictor fast enough to be used in practice.

We find that both decision trees and random forests based on light-weight Git features can successfully predict conflicts for different programming languages. Our observations shows that decision trees outperform random forests by an small margin. However, the f1-score of the safe class is much higher than the conflicting class.

Chapter 8

Threats to Validity

8.1 Internal Validity

`git merge` can use several merging algorithms and the choice of the algorithm used may impact the results. We employ the default one (recursive merging strategy) since developers typically do not change the default configuration of Git merge.

Rebasing is another strategy for integrating changes from different branches. When `git rebase` is used instead of `git merge` or when the `--rebase` option is used while pulling, a linear history is created and no explicit merge commits will exist. Therefore, there is a chance that we miss some merge scenarios since we detect merge scenarios based on the number of parents of a commit. Unfortunately, there is no precise methodology to extract rebased merge scenarios since there is no information in Git about them.

We eliminate n-way (octopus) merging and only focus on 3-way merging where each merge commit has exactly two parents. This may eliminate some merge scenarios. However, 3-way merging happens more often in practice.

We use a set of ranges for the random search of hyper-parameters. We selected these ranges based on our intuition and the heuristics in the literature about the hyper-parameters for machine learning techniques. However, we cannot guarantee that we found the globally optimal values for our hyper-parameters.

8.2 External Validity

While we have a large-scale empirical study, our evaluation is still limited to 105 open-source repositories on GitHub in seven popular programming languages. Our results may not address merge conflict prediction in other programming languages. Also, we need to train a separate predictor for each repository, and the effectiveness of a predictor that is trained on one repository should be investigated using other repositories.

Chapter 9

Implications and Discussion

Here we discuss what our prediction results may mean for avoiding complex merge conflicts in practice.

The recall of our decision trees is relatively high (0.79 to 0.92) for the conflicting class, which means that the classifier can identify an acceptable portion of conflicts if it is used as a replacement of speculative merging altogether. Notifying developers of these potential conflicts would allow them to merge early and avoid the conflict from becoming more complex. The downside is that the precision of predicting conflicts is lower (0.35 to 0.61), which means that developers may perform a merge earlier than needed (*i.e.*, perform a merge when there is no conflict to resolve). In practice, this may not be a big problem since frequent merges are encouraged to avoid conflicts in the long term.

However, instead of completely replacing speculative merging and running the risk of false positive notifications to developers, we advocate for using a merge-conflict predictor as a pre-filtering step for speculative merging [16], [17] or continuous merging [32] in developers' work environments (*e.g.*, their IDE). Both recall and precision of our classifiers for safe merges are considerably high (recall between 0.89 to 0.93 and precision in the range of 0.97 to 0.99 for decision trees). The precision of safe merge scenarios in the context of pre-filtering them out from speculative merging is important since we want to make sure that eliminated merge scenarios are actually safe. Given the high precision and the fact that conflict rates are typically low (10.0%), this means

that a subsequent proactive conflict detection tool will accurately eliminate a large number of safe merge scenarios from its analysis, thus potentially saving costs.

Chapter 10

Conclusion

In this thesis, We investigate if textual merge conflicts are predictable. If merge conflicts are predictable using inexpensive-to-extract Git features, we potentially can eliminate the safe merge scenarios in speculative merging and reduce the computational costs. We aim to train a binary classifier for each Git repository to be able to label unseen merge scenarios in the same repositories either as conflicting or not conflicting (safe) merge scenarios.

We develop a data gathering tool (MergAnser) to help us, and other researchers who work on software merging, to collect the data of direct merging and pull-requests in Git history. We use MergAnser to extract 147,967 merge scenarios from 105 popular and well-engineered open-source GitHub repositories in seven programming languages (C, C#, C++, Java, PHP, Python, and Ruby).

We extract 9 feature sets from each merge scenario to represent each of the merges as a fixed-length real-valued vector. We use the feature sets that are used in previous studies on software merging. We perform two studies using this data.

In the first study, we calculate the Spearman's rank-order correlation between conflicts and each of 9 feature sets. We found that there is no statistically strong correlation between our feature sets and conflicts. However, the feature set #1 (No. of simultaneously changed files in two branches) has medium and feature set #2 (No. of commits between the ancestor and the last commit in a branch) has weak correlations. Furthermore, only feature set #1 is an

important feature for decision trees that are trained using all feature sets.

In the second study, we focus on training binary classifiers to predict conflicts. For each repository we sort the available merge scenarios chronologically and use the first 70% for training and 30% for testing. Since the data is highly imbalanced and conflicts occur only in 10.0% of merge scenarios, we use cost-sensitive decision trees and ensemble learning models (random forests) to be able to handle this imbalanced data. The f1-score of our decision tree models, which outperform random forests by small margins, for predicting the not conflicting merges varies from 0.93 to 0.95 for different programming languages. This means that our models can effectively detect the safe merges that can be eliminated in speculative merging. The recall of conflicting merges also varies from 0.79 to 0.92, which indicates that our models are reliable to detect conflicting merges. However, the precision of our models is low (0.35 to 0.61) for the conflicting class which means that they might suggest unnecessary early merging which is not harmful to the development process, compared with the costs of resolving conflicts.

As the future step, we suggest investigating the impact of using our conflict predictors in speculative merging. The focus of this thesis is textual conflicts which are the conflicts that are reported by Git. As a future step, we suggest to investigate if predicting conflicts in structured merges is possible in practice. Due to the computational cost of structured merges, this predicting can dramatically reduce the costs of using structured merges.

References

[1] `https://github.com/Prickett/gitminer`.

[2] `https://chaoss.github.io/grimoirelab`.

[3] `https://github.com/ualberta-smr/merganser`.

[4] `https://github.com/ualberta-smr/merganser/wiki/`.

[5] `https://scikit-learn.org/`.

[6] P. Accioly, P. Borba, and G. Cavalcanti, "Understanding semi-structured merge conflict characteristics in open-source java projects," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2051–2085, 2018.

[7] P. Accioly, P. Borba, L. Silva, and G. Cavalcanti, "Analyzing conflict predictors in open-source java projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ACM, 2018, pp. 576–586.

[8] C. C. Aggarwal, *Data classification: algorithms and applications*. CRC press, 2014.

[9] T. W. Anderson and J. D. Finn, *The new statistical analysis of data*. Springer Science & Business Media, 2012.

[10] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 international conference on software engineering*, IEEE Press, 2013, pp. 712–721.

[11] J. Baumgartner, R. Kanzelman, H. Mony, and V. Paruthi, *Incremental speculative merging*, US Patent 7,934,180, 2011.

[12] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[13] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, IEEE, 2009, pp. 1–10.

[14] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ACM, 2012, p. 45.

[15] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[16] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ACM, 2011, pp. 168–178.

[17] ——, "Early detection of collaboration conflicts and risks," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1358–1375, 2013.

[18] J. Buffenbarger, "Syntactic software merging," in *Software Configuration Management*, Springer, 1995, pp. 153–172.

[19] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 59, 2017.

[20] O. Chapelle, B. Scholkopf, and A. Zien, "Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]," *IEEE Transactions on Neural Networks*, vol. 20, no. 3, pp. 542–542, 2009.

[21] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[22] C. Costa, J. Figueiredo, L. Murta, and A. Sarma, "Tipmerge: Recommending experts for integrating changes across branches," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2016, pp. 523–534.

[23] A. D. Da Cunha and D. Greathead, "Does personality matter?: An analysis of code-review ability," *Communications of the ACM*, vol. 50, no. 5, pp. 109–112, 2007.

[24] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen, "Effective software merging in the presence of object-oriented refactorings," *IEEE Transactions on Software Engineering*, vol. 34, no. 3, pp. 321–335, 2008.

[25] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, 2013, pp. 422–431.

[26] S. Ertekin, J. Huang, L. Bottou, and L. Giles, "Learning on the border: Active learning in imbalanced data classification," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, ACM, 2007, pp. 127–136.

[27] H. C. Estler, M. Nordio, C. A. Furia, and B. Meyer, "Awareness and merge conflicts in distributed software development," in *Global Software Engineering (ICGSE), 2014 IEEE 9th International Conference on*, IEEE, 2014, pp. 26–35.

[28] Y. Fan, X. Xia, D. Lo, and S. Li, "Early prediction of merged code changes to prioritize reviewing tasks," *Empirical Software Engineering*, pp. 1–48, 2018.

[29] *Fstmerge tool*, `https://github.com/joliebig/featurehouse/tree/master/fstmerge`.

[30] G. Gan, C. Ma, and J. Wu, *Data clustering: theory, algorithms, and applications*. Siam, 2007, vol. 20.

[31] G. Gousios, "The ghtorent dataset and tool suite," in *Proceedings of the 10th working conference on mining software repositories*, IEEE Press, 2013, pp. 233–236.

[32] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in *Proceedings of the 34th International Conference on Software Engineering*, IEEE Press, 2012, pp. 342–352.

[33] H. Guo and H. L. Viktor, "Learning from imbalanced data sets with boosting and data generation: The databoost-im approach," *ACM Sigkdd Explorations Newsletter*, vol. 6, no. 1, pp. 30–39, 2004.

[34] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, ACM, 2010, pp. 235–238.

[35] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review," *ACM computing surveys (CSUR)*, vol. 31, no. 3, pp. 264–323, 1999.

[36] *Jdime tool*, `http://fosd.net/JDime`.

[37] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[38] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*, ACM, 2014, pp. 92–101.

[39] ——, "An in-depth study of the promises and perils of mining github," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, 2016.

[40] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, 2013, pp. 732–741.

[41] M. G. Kendall, S. F. Kendall, and B. B. Smith, "The distribution of spearman's coefficient of rank correlation in a universe in which all rankings occur an equal number of times," *Biometrika*, pp. 251–273, 1939.

[42] T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano, "Comparing boosting and bagging techniques with noisy and imbalanced data," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 41, no. 3, pp. 552–568, 2010.

[43] O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, and B. de Water, "Studying pull request merges: A case study of shopify's active merchant," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ACM, 2018, pp. 124–133.

[44] H. Laurent and R. L. Rivest, "Constructing optimal binary decision trees is np-complete," *Information processing letters*, vol. 5, no. 1, pp. 15–17, 1976.

[45] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen, "Indicators for merge conflicts in the wild: Survey and empirical study," *Automated Software Engineering*, vol. 25, no. 2, pp. 279–313, 2018.

[46] A. Liaw, M. Wiener, *et al.*, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[47] D. Lindley, "Regression and correlation analysis," in *Time Series and Statistics*, Springer, 1990, pp. 237–243.

[48] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[49] M. Mahmoudi, S. Nadi, and N. Tsantalis, "Are refactorings to blame? an empirical study of refactorings in merge conflicts," in *Proc. of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*, 2019.

[50] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.

[51] S. McKee, N. Nelson, A. Sarma, and D. Dig, "Software practitioner perspectives on merge conflicts and resolutions," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, IEEE, 2017, pp. 467–478.

[52] G. G. L. Menezes, L. G. P. Murta, M. O. Barros, and A. Van Der Hoek, "On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by GitHub," *IEEE Transactions on Software Engineering*, 2018.

[53] T. Mens, "A state-of-the-art survey on software merging," *IEEE transactions on software engineering*, vol. 28, no. 5, pp. 449–462, 2002.

[54] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[55] H. T. Nguyen and A. Smeulders, "Active learning using pre-clustering," in *Proceedings of the twenty-first international conference on Machine learning*, ACM, 2004, p. 79.

[56] Y. Nishimura and K. Maruyama, "Supporting merge conflict resolution by using fine-grained code change history," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, IEEE, vol. 1, 2016, pp. 661–664.

[57] M. Owhadi-Kareshk and S. Nadi, "Scalable software merging studies with merganser," in *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*, 2019.

[58] M. Owhadi-Kareshk, S. Nadi, and J. Rubin, "Predicting merge conflicts in collaborative software development," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE, 2019, pp. 1–11.

[59] M. Owhadi-Kareshk, Y. Sedaghat, and M.-R. Akbarzadeh-T, "Pre-training of an artificial neural network for software fault prediction," in *2017 7th International Conference on Computer and Knowledge Engineering (ICCKE)*, IEEE, 2017, pp. 223–228.

[60] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[61] ——, "Simplifying decision trees," *International journal of man-machine studies*, vol. 27, no. 3, pp. 221–234, 1987.

[62] *Reaper dataset*, `https://reporeapers.github.io/static/downloads/dataset.csv.gz`.

[63] J. Reunanen, "Overfitting in making comparisons between variable selection methods," *Journal of Machine Learning Research*, vol. 3, no. Mar, pp. 1371–1382, 2003.

[64] A. Sarma, D. F. Redmiles, and A. Van Der Hoek, "Palantir: Early detection of development conflicts arising from parallel code changes," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 889–908, 2012.

[65] C. Schaffer, "Selecting a classification method by cross-validation," *Machine Learning*, vol. 13, no. 1, pp. 135–143, 1993.

[66] B. Scholkopf and A. J. Smola, *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2001.

[67] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2018, pp. 908–911.

[68] C. Strobl, A.-L. Boulesteix, A. Zeileis, and T. Hothorn, "Bias in random forest variable importance measures: Illustrations, sources and a solution," *BMC bioinformatics*, vol. 8, no. 1, p. 25, 2007.

[69] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *Proceedings of the 2006 symposium on Eye tracking research & applications*, ACM, 2006, pp. 133–140.

[70] J. Vesanto, "Som-based data visualization methods," *Intelligent data analysis*, vol. 3, no. 2, pp. 111–126, 1999.

[71] B. Westfechtel, "Structure-oriented merging of revisions of software documents," in *Proceedings of the 3rd international workshop on Software configuration management*, ACM, 1991, pp. 68–79.

[72] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?" *Information and Software Technology*, vol. 74, pp. 204–218, 2016.