University of Passau
Department of Informatics and Mathematics

Master's Thesis

GITCoP: A Machine Learning Based Approach to Predicting Merge Conflicts from Repository Metadata

Thomas Ziegler

January 20, 2017

Master's Thesis
at the Chair of Software Engineering
at the Department of Informatics and Mathematics
at the University of Passau

Instructor: Olaf Leßenich
Supervisors: Prof. Dr.-Ing. Sven Apel
Prof. Dr.-Ing. Norbert Siegmund

Abstract

Version Control Systems are a crucial tool in modern software development processes. They offer a number of features supporting the collaboration of multiple developers on a common code base. An important feature is branching which offers the possibility to start at a certain point in the version history of a project, make a copy of the entire code base, and work on this copy without changing the content of other branches. Those branches can be merged to combine all changes. If the same line in a file differs in two branches, a conflict during merging will occur. It is often the case, that two branches are constantly changed in parallel over a long period of time. Resolving the resulting conflicts can become tedious and time-consuming.
To overcome this problem, our goal is to predict potential merge conflicts between branches in advance, so that countermeasures such as early merging can be filed. To this end, we will use machine learning to learn important properties and features of a branch that are likely to indicate possible conflicts. Moreover, we propose tool support to predict conflicts in repositories in real time. Our results show that certain features, such as the number of simultaneously changed files and the number of commits on a branch are good indicators for conflicts. However, we also observe that there is no perfect prediction accuracy possible.

Contents

List of Tables

List of Figures

1. Introduction

In this chapter, we give an introduction to our work and define the aims of this thesis.

Modern software systems consist of millions of lines of code and are developed by large teams of developers over a long period of time. The complex process of developing software demands tools to support and manage it. One type are Version Control Systems (VCS). Depending on the system, they provide various features. Some features all common VCS support are

- **sharing** (every developer can provide changes for others to download),
- **versioning** (the system offers the possibility to retrieve older versions of the software),
- **branching** (multiple variants can be developed in parallel without interfering in other variants) and
- **merging** (variants can be merged together automatically).

When multiple developers collaborate using a VCS, usually every developer has a local copy of the code, to work on. When one of them reaches a point where she wants to share her current status with her coworkers, she publishes the changes to a shared repository or server. Another developer can pull the changes, which are then applied to her local copy. If someone does not like the changes, they can go back to the version before they were published and create a branch to continue working without them. When the problems are resolved, the branch can be merged to reflect all changes that were made.
Branches are also useful to develop new features. Usually one wants a "clean" codebase which only contains stable features, so untested code of new features should not be contained. But developing a feature can also be a process, people want to collaborate on. To maintain a clean codebase and share untested code at the same time, usually a new branch is created when the development of a new feature starts. Then the feature is developed and the code is published to that branch. When the feature is stable and the developers agree, that it should be in the stable codebase, the branch is merged.
There are other reasons to separate branches originating from the same code base. For instance, software product lines (SPL) are often developed using VCS. Each branch corresponds to an individual or a set of features of the SPL. Once a feature is implemented, it has to be introduced to the SPL to enable its combination with other features. This process is also a possible source of merge conflicts.
Those features to manage code and support collaboration make VCS are a crucial part of modern software development processes as they enable teams to collaborate on developing and maintaining giant codebases.

Even with tool support, multi-developer software engineering processes pose multiple challenges and some of them can not be overcome by using tools. While sharing and versioning are provided by VCS, merging of branches can only be automated under certain preconditions. While it is easy to merge two branches when no file has been modified by both of them, merging changes in the same file is more difficult. To automate this, most VCS use line-based merging. They take the changes of each branch and merge them by applying them line by line. This only works, if the same line has not been changed by both branches. In this case, the branches can not be merged automatically, a so called "*merge conflict*" appears.
Apart from those obvious conflicts, "hidden" conflicts can arise. For example, one developer changes the name of a function in his branch and another one uses the old name of the function. While the other type of conflict will be reported by the VCS, this type will only become visible when compiling the code fails or bug appear in the software.
Merge conflicts are a considerable problem in modern software development processes, especially when merges become huge. Resolving a few conflicts may be annoying, but when a feature branch, which was developed for months and adds thousands of lines of code, is merged, a huge amount of conflicts can arise and resolving them manually can take days or even weeks. For that reason, conflicts should be resolved when they are small and easy to understand or better, before they even occur.

In their paper "Indicators for Merge Conflicts in the Wild: Survey and Empirical Study" [1], Leßenich et al. tried to find out if the occurrence of merge conflicts is connected to certain parameters like the number of commits on the branches. They formulated seven hypotheses about indicators and merge conflicts, but were not able to proof any of them. So obviously, using single indicators to predict merge conflict does not result in satisfying results.

In this thesis, we try to find complex connections between different indicators and merge conflicts using machine learning. To accomplish this, we analyzed GIT repositories to extract features. We tested and tuned different machine learning algorithms and evaluated the results. As most merges do not produce conflicts, we had to take care of the class imbalance.
We used two approaches to predict merge conflicts. One is to pick two branches, extract the features and try to predict the conflict, we call it "multi-branch prediction". The other approach is to only look at one branch and find a way to predict the probability this branch will create a conflict, independent of the branch it is merged with. We call this "single-branch prediction".

In this thesis, we show that machine learning can be used to predict merge conflicts, but the accuracy of the prediction depends on the language and feature set you use. The best result on a dataset based on repositories containing C code were achieved by the random forest classifier. In the best case, it classified 94.8 % of all samples correctly. The results for the Java datasets were not as good, but still accuracy scores of 91.4 % and 90.7 % were achieved, also by the random forest classifier.
We describe the feature sets we use and show, that the number of files simultaneously changed in both branches is the most important feature to predict merge conflicts.
Additionally we introduce GITCoP, a tool to collect data from repositories and predict merge conflicts in realtime. It features a cache to store the data, so the

monitored repository does not have to be analyzed every time a prediction has to be made.
We evaluated the performance of the cache and showed, that initializing the cache takes some time, but the data necessary for predicting a merge conflict, can be retrieved in milliseconds. Even for huge repositories and a high number of branches, the cache fits in the memory of todays computers.

2. Background

In this chapter, we introduce different tools and techniques we use in this work.
There are two VCS commonly used in open source projects: Subversion (SVN) and GIT. We focus on GIT but all concepts are also applicable to SVN.
There are multiple ways of hosting a GIT repository. We use GitHub[1] to extract and analyze repository data, because it is open to public and one of the most popular hosting sites.
As said before, we use machine learning to predict merge conflicts. We explain the fundamentals of the approaches used in our tool and evaluation and we introduce scikit-learn, the machine learning library we used.

2.1. GIT

GIT is a distributed VCS meaning that there is no central repository like a "GIT Server". Each copy of the repository is a fully functional VCS, collaborating is done by pushing changes to and pulling changes from a shared repository.
Internally, the data is stored as blobs, that are identified by SHA-1 hashes and linked in the order of their creation.
What we refer to as "GIT" are on one hand the low-level ("plumbing") commands that handle the internal data structure and on the other hand the high-level ("porcelain") commands that use the low-level commands to provide advanced features. So, GIT is basically a bunch of tools but in this thesis we refer to it as one piece of software including all commands and the data structure.
The documentation of GIT can be found at `https://git-scm.com/doc`. In the following subsections, we will discuss those parts of GIT relevant for this work.

2.1.1. Commits

After a developer made changes to the code in a repository, she can make them known to GIT. This is called *commiting*. A commit is a set of changes which is stored in an atomic way. It is not possible to apply only part of the changes in a commit.
As said before, GIT stores the data in blobs which are linked. There is always exactly one blob for a commit which is linked to the commit that happened immediately before, thus creating a chain of commits.

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

2.1.2. Branches

GIT and most other VCS provide techniques to maintain and develop multiple variants of code at the same time in the same repository. This is called *branching*. A new branch can be created at any commit in the repository. A commit is always at a single branch while all other branches in the repository are unaffected by it.
When a new branch is created a new child is added to the blob which contains the parent commit in the internal representation. Each branch has a pointer to the last commit on this branch, so all information we have about branches are their names and where they end. There is no information about where or when the branch started so internally every branch starts at the "root" of the repository, the initial commit.
As every commit has to be on one branch, the initial commit in a repository creates the first branch. It is called the `master` branch and is often used in software development to carry stable versions of the code.

2.1.3. Merges and Conflicts

In most software development processes, it is common to develop new features on a separate branch such that the development of other features and the stability of the existing code is not affected. Once a feature is considered to be stable, one wants to have it in the project, so the code has to get into the stable branch. Instead of manually copying the code from the development branch, we can *merge* the content of one branch into another one.
In GIT, there are two main ways to do this. Assume there is a branch called `stable` and a branch called `feature`. The `feature` branch has been split off from `stable` and contains some new commits that should now be merged back.
In the simple case when no commits were made on `stable` after `feature` has been created, a so called *fast-forward merge* can be applied. That is, the new commits on `feature` are simply applied on top of `stable`.
If a fast-forward merge is not possible, GIT tries to combine the different versions of the files in both branches. After this is done the developer who initiated the merge can make further changes and then commit the merge. This results in a so called *merge commit* which has two ancestors.
If there are no files that have been modified in both branches, merging is easy as only the newest versions of each file has to be used, but if the same file has been modified in both branches the corresponding changes have to be combined. GIT ships a tool that can do such a combination automatically as long as the changes are in different lines. If a line in a file has been modified in both branches, the tool does not know which one to take and a so called *merge conflict* occurs. This causes the merge process to halt and the developer has to resolve the conflict by manually selecting the correct version or by implementing a custom fix. Afterwards, she can commit the merge.
In the merge commit, GIT stores which commits have been merged into the branch. To determine which commits belong to a merge commit, GIT looks for the last commit that is present in both branches, this commit is called *merge base*. All commits that occurred on the branch that is to be merged after the merge base have to be applied in the merge commit.

It is also possible to merge more than two branches at once. This is called an *octopus merge* and the merge commit has as many ancestors as the number of branches which were merged. In this work, we do not consider octopus merges as the same result can be achieved by merging each branch in a separate commit.

2.1.4. Scenario

In this thesis, we use the term *(merge) scenario* for a tuple (F, C) where F is a set of features from two branches, that are merged, and C is the information if a conflict appeared while merging the branches. Potential features of F are explained in Section 3.1.2. We collect these features from a number of commits, usually those that occurred after the youngest common commit of both branches. A scenario can also just include features from one branch.

2.2. GitHub

GitHub is a popular platform for hosting open source projects and has countless public source code repositories. It provides access to the repositories via HTTPS and SSH and features a REST API to access meta data of users, organizations, and repositories. We used the API to obtain a list of repositories to analyze and evaluate our classifiers. The documentation of the API can be found at `https://developer.github.com`.

2.3. Machine Learning

Machine learning is the branch of Computer Science which focuses on finding algorithms that enable computers to make decisions about the real wold.
In this section, we introduce a few common terms and concepts of machine learning as well as the library we used in our implementation.
Machine learning can be categorized into supervised and unsupervised learning. In unsupervised learning, we feed unlabeled data to an algorithm that tries to find hidden structures. Clustering, for example, tries to group data-points based on similarities. In supervised learning, an algorithm is trained using labeled data before it has to classify unlabeled data.
In our case, a merge scenario is a data-point and the property “conflict” respectively “no-conflict” is the label, which makes the conflict prediction problem a classic classification problem.
The classification task begins with a process called fitting, in which a classifier learns weights on given input parameter (i.e., features) to predict the output class (i.e., the label) of an observation (i.e., a merge scenario). Afterwards, the classifier contains a model which is then used to predict the label of new unlabeled data. Some classifiers are also able to provide the probability of an observation to be in one specific class. Additionally to classification, there is regression, where the aim is to predict a numeric value instead of a label.
In supervised learning, we can evaluate a classifier by using a subset of the whole

data to train the classifier (training set) and giving the remaining set without the labels to the classifier to evaluate the classification accuracy (test set). Afterwards, we compare the labels the classifier predicted with the actual labels. The more data points were classified correctly, the better the classifier is. The methods used to evaluate the classifiers will be discussed in Chapter 4.
A common problem in supervised learning is *overfitting*, which describes the phenomenon when the model of a classifier fits the training data very precisely but fails to classify data points which are not contained in the training set. That is, the classifier did not generalize over the training set. Some classifiers are more prone to overfitting than others. They have to be tuned to avoid this. [2]

2.3.1. scikit-learn

Scikit-learn[2] is a free machine learning library written in Python[3]. It is one of the most popular free libraries for this purpose and features a broad documentation which can be found at `http://scikit-learn.org/0.17/documentation.html`. We decided to use scikit-learn (version 0.17) because it has all features we require and provides results without excessive coding overhead.
The library contains implementations of a number of machine learning algorithms as well as tools for preprocessing data and evaluation tasks. We will discuss those in detail when they are used. [3]

2.3.2. Algorithms

There are multiple algorithms for supervised learning with different strengths and weaknesses. Amongst them are Support Vector Machines (SVMs), Decision Trees and Naive Bayes. In the following subsections, we will shortly describe the algorithms relevant for this work and discuss their strengths and weaknesses.

Decision Trees

Decision trees try to classify data by "asking questions" about the item to classify. For example, assume there is a zoo and a database containing the animals in the zoo including information about the species and whether it is viviparous.
The animals are to be classified into mammals and non-mammals. The first question could be if the animal is viviparous. If it is, it is a mammal, if it is not the second question could be about the species. If it is a platypus, it is a mammal, if not, it is not a mammal. In Figure 2.1 we depict the corresponding decision tree. The rectangular boxes are "question nodes". If the questions is answered with "yes", the next step is the left child, else it is the right one. Leaf nodes are depicted as ellipses and contain the decision the classifier comes to, when it reaches the node.
During training, the tree containing the questions and the resulting labels is created. There are multiple algorithms to do that but all of them try to find the feature (i.e., split or question) that provides the greatest information gain if you split the data on

[2] `http://scikit-learn.org`
[3] `https://www.python.org/`

it. Then, the data is split and for each group the process is repeated. This continues until all groups contain only items with the same label or a certain threshold is reached. Scikit-learn uses an optimized version of the `CART` algorithm that can also be used for regression, not only for classification.[4]
An advantage of decision trees is that they are computationally cheap and can deal with irrelevant features and missing values, but they are also prone to overfitting which can be mitigated by tuning different parameters. This will be discussed in detail in Section 3.4. [4]

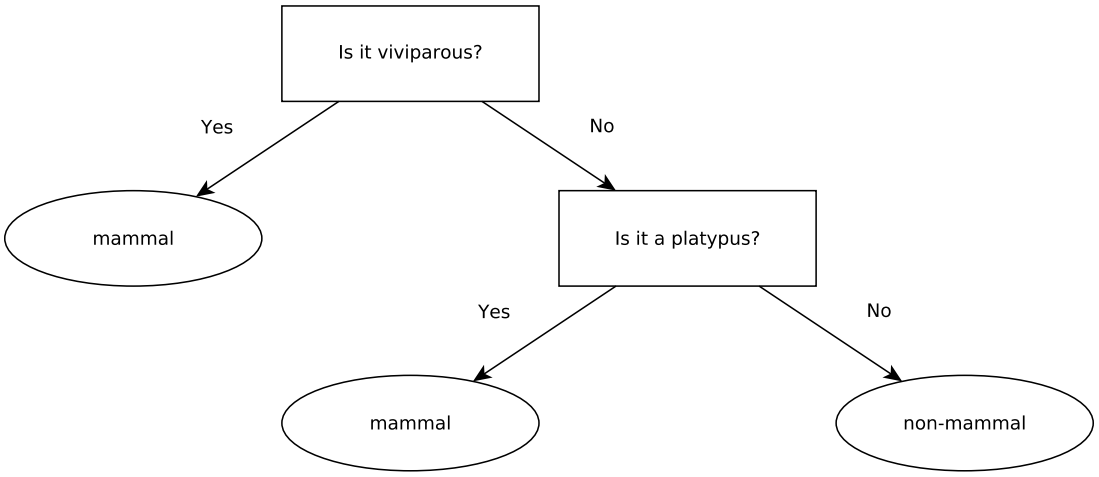

Figure 2.1.: A decision tree to determine whether an animal is a mammal.

Support Vector Machines

Support vector machines separate the items in a dataset using a separating hyperplane. The plane is chosen in a way that the closest items of each group are as far away from the plane as possible. Those items are called *support vectors*.
Most datasets are not shaped in a way that the groups can be separated by a hyperplane in a satisfying way. To mitigate this problem, so called *kernels* can be used to transform the data to a higher-dimensional feature space where the separation is possible. [4, 5]
Scikit-learn uses `libsvm`[5] and `liblinear`[6] to provide SVMs.
The documentation of scikit-learn claims, that "the fit time complexity is more than quadratic with the number of samples which makes it hard to scale to dataset with more than a couple of 10 000 samples."[7] [8] This is a huge drawback as we use datasets that contain more than 100 000 samples. Further descriptions of the datasets can be found in Section 3.1.

[4] `http://scikit-learn.org/0.17/modules/tree.html`
[5] `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`
[6] `http://www.csie.ntu.edu.tw/~cjlin/liblinear/`
[7] `http://scikit-learn.org/0.17/modules/generated/sklearn.svm.SVC.html`
[8] `http://scikit-learn.org/0.17/modules/svm.html#complexity`

Naive Bayes

Naive Bayes classifiers are based on Bayes' theorem and commonly used in text classification. This method assumes that all features are independent of each other and contribute to the probability, that a sample is in a specific class, in equal amounts. During training, the classifier collects all possible values for each feature and calculates the probability per class for each value. When classifying an unknown sample, it takes each feature and determines the probability for each class. The reported probability for this sample to be in a specific class is the average of all its features for this class. [4, 6]
The classifier can use different probability distributions. Scikit-learn ships three such classifiers, the gaussian-, multinomial- and bernoulli naive bayes classifiers.[9] Details about the probability distributions can be found in "Pattern Recognition and Machine Learning" by C. Bishop. [5]

Logistic Regression

Logistic regression uses a generalized linear model (GLM) to predict the probability of a sample belonging to a class. Describing GLMs and linear regression in detail would be beyond the scope of this thesis. Further information can be found in [5] and on the scikit-learn website[10]. Here we will just give a short description.
Logistic regression is, despite its name, a method for classification. Its strength lies in providing probabilities for a sample to belong to a class and not just the class label. In its basic form, it is only capable of binary classification. As our problem is binary, we will not describe how a multi-class logistic regression classifier works. Before explaining logistic regression, we have to define the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.1}$$

The sigmoid function is 0.5 for $z = 0$, approaches 1 for increasing values and 0 for decreasing values.
We assume all our samples contain n features. The features of a sample s are denoted as $f_s^0 \dots f_s^{n-1}$. Additionally, for each feature f^i, with $i \in [0, n[$ exists a weight X^i. The probability $p^C(s)$ of a sample s to belong to class C is calculated as follows:

$$p^C(s) = \sigma\left(\sum_{i=0}^{i<n} f_s^i X^i\right) \tag{2.2}$$

So a logistic regression classifier multiplies each feature with its weight and sums them up. Then the sigmoid of the sum is calculated and the result is the probability for the sample to belong to class C. If $p^C(s) > 0.5$ the sample s is classified as C, for $p^C(s) < 0.5$ it will be put in the other class.
During training of a logistic regression classifier, the optimal weight X^i for each feature f^i is determined. This is an optimization problem which can be solved in different ways. [4]

[9] `http://scikit-learn.org/0.17/modules/naive_bayes.html`
[10] `http://scikit-learn.org/0.17/modules/linear_model.html`

2.3.3. Ensemble Methods

In ensemble methods, multiple classifiers are combined to improve the accuracy. In this thesis, we use two types of ensemble methods: *Bagging* and *Boosting*. [7, 8]
Bagging methods combine predictions of their classifiers and returns the average of all. Usually, they are used with decision trees, but this method can be applied to any machine learning algorithm that is capable of predicting the probability for a sample to be in a class. Bagging methods were introduced by Leo Breitman in his article "Bagging Predictors". [9]
There are multiple approaches to Boosting. All have in common that they do not consider the predictions of all classifiers equally but apply different weights or train different classifiers to perform well on different subsets of the training data. [10, 11]
Scikit-learn provides implementation of different ensemble methods[11] [12]. We use the RandomForest[13] and the AdaBoost[14] classifiers which are described in the following subsections. [3]

Random Forest

Random forest is a bagging method, which uses multiple decision trees that are trained using a random subset of the training set. Also each split is not the best split among all features, but the best split among a random subset of the features. Initially, it was proposed to let each classifier vote on a class and take the class with the most votes as the final result, but the implementation of scikit-learn averages the probabilities reported by all trees and uses it to get to a final decision[15]. [11, 12]

AdaBoost

As indicated by the name, the AdaBoost algorithm is a boosting method. It is an iterative approach where at each iteration the weight of those samples, that were put in the wrong class in the previous iteration, is increased so the next training step focuses more on those. Hence, a new classifier is trained in each iteration. When classifying an unknown sample, the weighted prediction of all classifiers in the ensemble are combined to come to a final prediction[16]. [13]

[11] http://scikit-learn.org/0.17/modules/ensemble.html
[12] http://scikit-learn.org/0.17/modules/classes.html#module-sklearn.ensemble
[13] http://scikit-learn.org/stable/0.17/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier
[14] http://scikit-learn.org/0.17/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier
[15] http://scikit-learn.org/stable/modules/ensemble.html
[16] http://scikit-learn.org/0.17/modules/ensemble.html

3. Implementation

The main goal of this work is to find an algorithm and the best predictive features to accurately predict whether a conflict occurs in a merge scenario. Moreover, we want to provide an accompanying tool that implements the techniques and is able to predict the probability of merge conflicts.
In this chapter, we describe the data we used to achieve this goal, show how different machine learning algorithms can applied and tuned, show different possible feature sets, and present GITCoP, our tool to predict merge conflicts.

3.1. Data

In this work, we use three sets of data. One is the data used by Leßenich et al. in [1][17]. It was collected for the aforementioned paper and merges were performed by JDime, a structured-merge tool. From here on, we refer to it as the *jdime-dataset*.
We built the other two datasets based on our tool. To this end, we first took the repositories from the jdime-dataset, which were still available on GitHub and analyzed them using the default GIT tools instead of JDime. From here on, we refer to it as the *git-java-dataset*. For the third dataset, we did the same but instead of using the repositories containing Java code, we used 200 repositories containing C code. We refer to it as the *git-c-dataset* from here on.
In the following two subsection, we will describe the datasets in detail. Afterwards, we list and describe relevant features.

3.1.1. Datasets

As said before, the **jdime-dataset** was created using JDime, a special merge tool for Java. In their paper, Leßenich et al. write, they used 32 579 merge scenarios from 155 open-source projects. We found that only 16 742 scenarios were useable for our purpose, as a lot of scenarios were inconsistent. For example, the original dataset contained scenarios, where nothing was changed on one of the branches.
The original dataset is also focused on Java and contains a lot of metrics provided by JDime. We want to create a method to predict merge conflicts for any language so we did not use any Java-specific features, that were present in the original dataset. For that reason we started mining our own dataset, the **git-java-dataset**, based on the 710 repositories listed in the original one. For that purpose we wrote a tool which also shares some code with the GITCoP cache.
We used it to analyze the repository and extracted a total of 108 368 merge scenarios

[17] `http://www.infosun.fim.uni-passau.de/spl/papers/conflict-prediction/`

including 14 437 ones where at least one conflict occurred. We found, that even from the data obtained with our tool, some of the scenarios had to be removed. As some unexpected situation occurred during analyzing some repositories, that should not occur with GIT, we suspect that some repositories were tempered with in the past. But most of the scenarios were sound.
We used our analyzer to create the **git-c-dataset** for which we analyzed the 200 repositories containing C code with the most stars on GitHub. From those we obtained 36 756 scenarios of which 7 218 contain conflicts.

In Section 3.1.2 we will show that the number of simultaneously changed files is a feature with special importance as if this number is zero, a conflict is not possible. For that reason, we used two flavours of each dataset. The "full" one, where all scenarios are included and the "reduced" one, where all scenarios are removed in which the number of simultaneously changed files is zero.
In Table 3.1 we show a summary of all datasets and flavours.

Table 3.1.: The number of data points in the datasets

		jdime-dataset	git-dataset	
		Java	Java	C
full	Total	16 742	108 368	36 756
	No conflict	15 134	93 934	29 538
	Conflict	1 608	14 434	7 218
reduced	Total	4 824	38 549	19 678
	No conflict	3 216	24 115	12 460
	Conflict	1 608	14 434	7 218

The total number of scenarios shows that the jdime-dataset has a very high rate of scenarios where conflicts are impossible (71.2 %) while it is a lot lower at the set containing C project (46.6 %). In the git-java-dataset the percentage (64.4 %) is higher but still not as high as with the jdime-dataset. One could assume that in Java projects, merges where no files have been changed simultaneously are more common that in C projects.
The percentage of scenarios containing a conflict is lowest in the jdime-datasets. In the full set it is only 9.6 % but in the reduced one it is 33.3 %. In the git-java-dataset it increases from 13.3 % to 37.4 % and in the git-c-dataset from 19.6 % to 36.6 % when removing the obviously not conflicting scenarios.

3.1.2. Features

In this subsection, we describe the features that are relevant for this work. Some of them are not available in the JDime-dataset, which we will state explicitly.
At first, we will list those features which can be computed for each branch separately. This is an important property of a feature, as it allows us to predict merge conflicts when only looking at a single branch. In feature sets for a scenario, those features are included twice, once for each branch.

The number of changed files in a branch.

The number of commits in a branch.

The number of developers which were active in a branch.

The average length of the commit messages in a branch. This is not available in the JDime-dataset.

The number of lines added to relevant files in all commits on the branch.

The number of lines deleted from relevant files in all commits on the branch.

The following features can only be computed once for a scenario and not for each branch separately.

The number of files changed simultaneously in both branches Files that exist in both branches and have been modified by both of them are considered a possible predictive feature. A conflict is impossible if this number is zero, so this feature might be highly relevant. Still it can be desirable to avoid using this feature, as it is quite expensive to compute and can not be computed for each branch separately.

The number of files containing conflicts in a merge scenario. This actually provides the label for the classes and is not used as a feature for classification.

In the GIT-datasets only changes in files of a relevant type are considered and commits are only considered if at least one file of a relevant type has been changed. For example, only the number of additions in relevant files are reported and if a developer only committed without ever changing a relevant file in his commits, he will not be counted in the "number of developers"-feature. Consequently, commits which do not change a relevant file are not counted in the "number of commits"-feature.
How this has been handled in the JDime-dataset could not be determined any more.

3.2. Feature Engineering

One main contribution of this work is to find one or more sets of features that can be used to precisely predict merge conflicts. Here, we present multiple possible sets of features including their pros and cons.

The simple feature set includes the number of changed files, commits, developers, lines added, and lines deleted for both branches. The features can be collected for a branch without directly comparing it to another one, which makes it easy and fast to compute. On the downside, the results are not as good as those achieved with the extended feature set.

The half feature set includes the same features as the simple set, but only for one branch. In theory, this has the advantage that one can predict the probability that a branch causes a conflict without, irrespectively of the other branch. To do that in a case, where the other branch is actually unknown, is not possible, as the point in the history, where the branch starts, is unknown and the features can not be computed. The result are also not as good as those achieved with the simple feature set.

The extended feature set includes all features of the simple one plus the number of simultaneously changed files. It turned out that this is the feature set with the most accurate predictions.

As mentioned in Section 3.1.2, the number of simultaneously changed files has special importance. Additionally to the fact that conflicts are impossible if it is zero, it is also the most important feature as adding it improves the accuracy of a model significantly.
In Figure 3.1 a tree model is shown where a decision tree classifier was fitted to the GIT-java-dataset using the features of the extended feature set. The maximal depth of the tree was set to 4 to reduce the size of the figure.
A box in the picture depicts a node of the tree, the first line is the name of the feature and the constraint on which a split happens. This line is not present, if the node is a leaf node. The child node for the samples where the constraint in the first line is true is located left of the parent node, the other child is always on the right side. The feature "FilesChangedInBoth" is the number of simultaneously changed files and "LeftAdditions" is the number of additions on one specific branch.
At the first node you can see, that the tree will classify any scenario where the number of simultaneously changed files is smaller than 0.5 as "no-conflict". As only integers can appear here, this means, that any scenario where the number is zero, will be put in the "no-conflict" class. The right child node splits based on this feature again. The fact that three of the four splits are based on the same feature, emphasizes its importance.

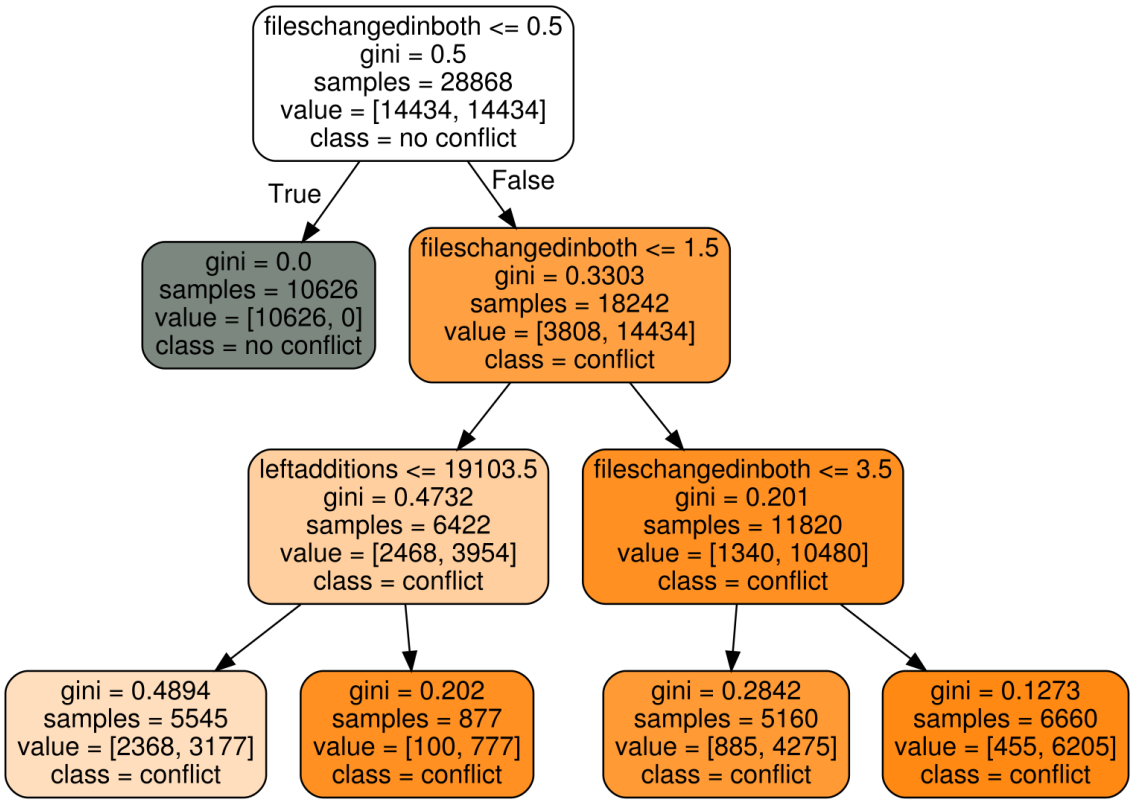

Figure 3.1.: Tree model created with the GIT-Java-dataset and the extended feature set.

3.3. Quality Measure for Branches

The quality of a branch in a repository can be measured by how many conflicts it produces when it is merged with other branches. But for calculating this, two branches are needed and it is impossible to determine which of them is responsible for the conflict.
In this section we introduce a quality measure for branches based on the probability a conflict occurs without using data from another branch or merging.

We define the quality measure $q_t^c(b)$ of a branch b to be the number n of youngest commits, that have to be in a scenario as defined in section 2.1.4 so the classifier c predicts the probability of a conflict to be greater or equal to t. The classifier c has to be trained using the "half feature set" described in section 3.2 as the other feature sets contain features which can only be calculated if there are two branches. It also has to provide the probability for a sample to belong to each of the classes.

We assume, we have a specific classifier C that complies to the specification above and there is a branch B. Furthermore we define $t = 0.8$.
To compute the quality measure $q_{0.8}^C(B)$ we take the youngest commit in B, retrieve the features defined in the "half feature set" and build a merge scenario from them. That we feed to the classifier and get the probability p for the scenario to produce a merge conflict. If $p >= t$, the quality measure $q_{0.8}^C(B) = 1$, else we add the next commit to the scenario and compute p again. This is repeated until $p >= t$ holds. $q_{0.8}^C(B)$ is the number of commits that have to be in the scenario to get a probability greater or equal to t. Pseudo code of the algorithm is also shown in Listing 3.1.

Listing 3.1: Computation of the quality measure in pseudo code.

```
C = Classifier()
L = repository.branch(B).commits()
t = 0.8
p = 0
q = 0

while p < t:
    q += 1
    if q >= len(L):
        throw OutOfCommitsException()
    p = C.predict_proba(features(L[:q]))

return q
```

3.4. Machine Learning Algorithms and Tuning

An important challenge of this work is to find a selection of algorithms that perform well on the classification problem and to tune each algorithm separately using its specific parameters. In this section, we will show different algorithms and possible tuning parameters. We also shortly describe their performance. The exact evaluation results can be found in Sections 5.2.1 and 5.2.4.

3.4.1. Decision Trees

Decision trees have been introduced in section 2.3.2. They appear to be the most flexible and in general best performing algorithm for our problem. Over all datasets and feature sets they proved to be the most accurate classifiers of non-ensemble methods.

To avoid overfitting, we restrained the growth of the tree. To this end, we limited the maximum depth of the tree and set a number of samples that have to be present in each leaf during training. The last one stops the algorithm from splitting a set of samples if the number of the samples in the resulting sets would be lower than a certain threshold.

Another common technique to avoid overfitting is *pruning*. It reduces the size of the tree by removing unimportant parts of the tree after training. Scikit-learn version 0.17 does not support pruning, hence we could not use it.[18]

The `DecisionTreeClassifier` is located in the `sklearn.tree` package. To create a new instance, we use code similar to that shown in Listing 3.2.

Listing 3.2: Creation of a new DecisionTreeClassifier instance.

```
1 dtc = DecisionTreeClassifier(max_depth=max_depth,
2                               min_samples_leaf=min_samples_leaf)
```

3.4.2. Support Vector Machines

The possibility to use different kernels makes them useful for a large variety of problems, but they are also highly configurable. Depending on the kernel, there are different parameters to tune.
The SVMs performance depends on the kernel as well, but in most cases, the most accurate SVM was still less precise than a decision tree.

A drawback of SVMs is, that the time required for training increases rapidly with the number of samples in the training set. Scikit-learn's documentation of SVMs[19] claims that, depending on some factors such as cache usage, the time-complexity of training is between $\mathcal{O}(n_{features} \times n_{samples}^2)$ and $\mathcal{O}(n_{features} \times n_{samples}^3)$. Where $n_{samples}$ is the number of samples in the training data and $n_{features}$ is the number of features of each vector.

Depending on the kernel, there are up to three parameters that influence the performance of the classifier. Scikit-learn ships four kernels for SVMs by default. Those are the

- **linear kernel** which tries to separate the classes linearly,
- **rbf kernel** which uses the radial bias function,
- **polynomial kernel** which uses a polynomial function and
- **the sigmoid kernel** which uses the sigmoid function.

[18] `http://scikit-learn.org/0.17/modules/tree.html`
[19] `http://scikit-learn.org/0.17/modules/svm.html`

A parameter all SVMs share, regardless of the kernel, is the penalty of the error term, called `C`. A higher value for `C` allows the SVM to take more samples as support vectors, fitting the model better to the training data, while lower values make the "decision surface more smooth"[20].
A kernel coefficient shared by the rbf, polynomial, and sigmoid kernel is called *gamma* and defines the influence of each sample. Both parameters, gamma and `C` have to be optimized at the same time, as their influence depends on the value of the respectively other parameter.
For the polynomial kernel, there is an additional parameter, called *deg*, which sets the degree of the polynomial function used by the kernel. [14]

We used the class `SVC`, which is located in the `sklearn.svm` package. To create instances of SVMs we used code similar to the one we show in Listing 3.3.

Listing 3.3: Creation of a SVM instance with the polynomial kernel.

```
1 svm = SVC(kernel='poly', cache_size=3000,
2           gamma=(10 ** gamma_log), C=C, degree=deg)
```

3.4.3. Naive Bayes

There are no parameters to tune for Naive Bayes classifiers, so they were used as provided by scikit-learn. Other than a few exceptions, they did not perform as well as the other classifiers.

The Naive Bayes classifiers are located in the package `sklearn.naive_bayes`. The constructors of the classes take no parameters. An instance of a Gaussian Naive Bayes classifier, for example, can be created like this: `gnb = GaussianNB()`.

3.4.4. Logistic Regression

To implement the quality measure as defined in section 3.3 we need methods that provide the probability for a sample to belong to a certain class. For this we tried using logistic regression as described in section 2.3.2. For the optimization problem we used the Stochastic Average Gradient descent solver provided by scikit-learn. The only parameter we optimized is the regularization strength which can be used to avoid overfitting.
The logistic regression classifier did not proof to be very accurate.

The implementation resides in the `LogisticRegression` class, which is located in the `sklearn.linear_model.logistic` package. Instances are created like this: `log = LogisticRegression(C=C)`.

3.4.5. Random Forest

The random forest method is based on decision trees but has different properties and additional tuning parameters. While it is necessary to limit the growth of a decision

[20] `http://scikit-learn.org/0.17/auto_examples/svm/plot_rbf_parameters.html`

tree to avoid overfitting, this is not the case with random forest classifiers because each tree in the forest is trained with a different set of samples. The fact, that each tree votes for a class during the classification process, leads to proper generalization over the complete training set.
But it can still be desirable to limit the growth of the tree to reduce training time and the amount of memory consumed by the model. In our case, the size of the models were reasonably small so we decided not to limit the depth of the trees.

As mentioned in section 2.3.3, usually not all features are considered while finding the split during training. The number of features in that subset (n_{split}) is one of the most important parameters to tune. By using the total number of features, this can be eliminated. Usual choices for the number of features are $n_{split} = \sqrt{n_{features}}$ or $n_{split} = \log_2(n_{features})$ where $n_{features}$ is the total amount of features.
We evaluated the random forest classifier using three different values for n_{split}: $\sqrt{n_{features}}$, $\log_2(n_{features})$ and $n_{features}$. Those are already predefined in the implementation, if all features should be used, `None` can be passed as the `max_features` parameter, for the other two options, the strings `'log2'` and `'sqrt'` can be used.

With an increasing number of trees in the forest, the accuracy of the classifier increases as well as the training time and memory consumption of the model. At some point, adding more trees does not notably improve the accuracy of the classifier any more. We used random forests with 400 trees as they yield good performance and keep training time and memory consumption to a reasonable.

The random forest classifier was the most accurate classifier we evaluated. Except for one case, it achieved the highest accuracy score. The only parameter, that was not fixed was the number of features to consider when splitting.

Scikit-learn's implementation supports training the trees in parallel. The number of processes to use can be defined by using the `n_jobs` parameter. Passing -1 will use as many threads as processors in the machine. The random forest classifier is implemented in the `RandomForestClassifier` class which is located in the `sklearn.ensemble` package. The code we used resembles that shown in Listing 3.4.

Listing 3.4: Creation of a random forest classifier.

```
1 rfst = RandomForestClassifier(n_estimators=400,
2                                max_depth=None,
3                                min_samples_split=1,
4                                max_features=max_features,
5                                n_jobs=-1)
```

3.4.6. AdaBoost

We tested multiple classifiers as base for the AdaBoost algorithm and decided to only use decision trees. When Naive Bayes were used, AdaBoost did hardly ever provide significant improvement, SVMs were discarded as base, as tuning would have been too time-consuming due to the complexity of SVMs during training.
The AdaBoost classifier with decision trees as base proofed to be one of the most accurate classifiers we used.
We tuned AdaBoost by modifying the number of estimators.

The implementation resides in class `AdaBoostClassifier`, which is located in the `sklearn.ensemble` package. We used code similar to that shown in Listing 3.5 to create instances.

Listing 3.5: Creation of an AdaBoost classifier with a decision tree as base.

```
ada = AdaBoostClassifier(
    base_estimator=DecisionTreeClassifier(
        max_depth=max_depth,
        min_samples_leaf=min_samples_leaf),
    n_estimators=n)
```

3.5. Monitoring Tool

In addition to finding an algorithm and feature sets to accurately predict merge conflicts, we implemented a accompanying tool to monitor a repository and warn the developer about merge conflicts in real-time.
Ideally, when a developer wants to commit a change, he will instantly be informed about a possible merge conflict if the corresponding probability is above a certain threshold. This poses the following challenge:
The probability of conflicts with all other branches has to be calculated. As the merge base is possibly different for each branch, we have to combine the active branch with each other branch and analyze the combination to get a feature set. So basically, we would have to simulate every possible merge with the active branch.
But retrieving the feature set is time consuming as one or more GIT commands have to be executed and the output has to be parsed and processed. So this way would not give us great advantage over actually doing the merge. As GIT has an option to abort the merge this could even be automated and would not alter the repository state.
The approach of merging-and-aborting as well as the one to analyze the full repository would be time-consuming and render the repository unusable for some time which would interfere with the developers workflow.

3.5.1. Caching

To mitigate those problems, we build a cache to store the parameters of a commit that we need to assemble the feature set. The cache enables us to retrieve the information of all possible merge scenarios without accessing the repository. The repository is only needed to create the cache for the first time and to update it, when new commits are added. New information has to be retrieved from the repository only once and is then persisted in the cache.
We use the cache to store and retrieve information, we need to predict merge conflicts, in a way that enables us to quickly retrieve merge scenarios. In fact, we are able to access all necessary information in constant time.

The cache consists of a linear history of every branch in a double-linked list where the head of the list contains the information about the latest commit. Each element is linked to a real commit in the repository and contains the accumulated information

of the corresponding commit and all commits that succeeded it. In Figure 3.2 an example of a cache containing information about three commits can be found. The commits are numbered in the order they occurred, so *commit 1* is the oldest one.

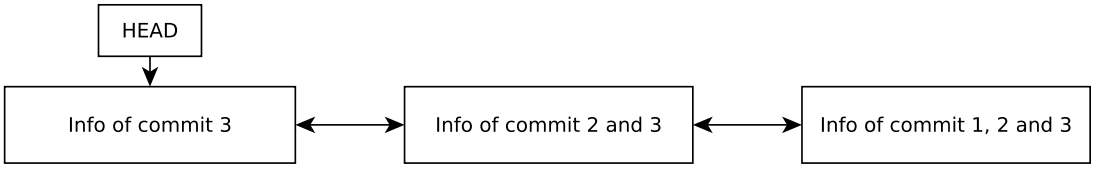

Figure 3.2.: Depicting a cache containing three commits.

If a new commit occurs on a branch, the commit has to be analyzed and the cache of the whole branch has to be updated as each element in the list reflects the changes that were made in the following commits. In case of an update, a node containing the information about the new commit is attached as new head of the list. Then, the already existing nodes are updated. Hence, an update costs $\mathcal{O}(n)$ with n being the number of elements in the cache. We show the process of updating the cache in Figure 3.3.

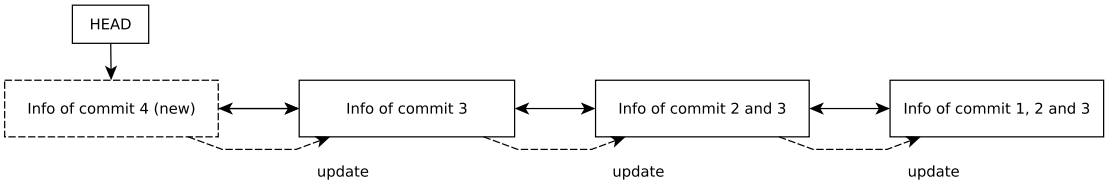

Figure 3.3.: Showing the process of updating the cache.

To provide random access to the cache, the implementation maintains a map of commit hashes to their corresponding entries in the cache using a python dictionary. This means that we can access any item in the cache in $\mathcal{O}(1)$ in the average case.

In the scenario where a developer commits and then wants to be informed about likely conflicts in the future, the information about the recent commit has to be extracted from the repository and then the cache of the current branch has to be updated. After that all the information needed to assemble the feature sets are already in memory, so the repository does not have to be analyzed and the cache does not have to be altered further. Hence, the feature sets of all relevant merge scenarios can be computed in $\mathcal{O}(k)$ where k is the number of branches.

As all branches start at the initial commit of a repository and repositories can get very huge over time[21] initializing the cache would take a huge amount of time and the cache requires a lot of memory. But a cache that contains all items is not necessary for various reasons:

- Two branches that are meant to be merged mostly have been diverged only few commits earlier. But only the commits that happened after the branches diverged are relevant for predicting merge conflicts.

[21] On 23rd of July 2016, the repository containing the linux source code had over 600 000 commits.

- Two branches that diverged hundreds of commits ago are very likely to cause conflicts when merged. In this case we do not have to compute a feature set and feed it to a classifier as we can assume that a conflict will occur with a very high probability. If such branches exists, they are often even not meant to be merged at all, for example, because they provide only support for some legacy version which is not actively supported anymore.

To prevent unnecessary information to be stored in the cache, it is possible to specify a list of branches which are monitored. Additionally, the tool will automatically compute the youngest common ancestor of all branches. This is the oldest commit in the repository that has to be represented in the cache as all older commits already exist in all branches. No commit older than this one will be present in the cache.

As said before, sometimes branches exist which diverged long ago and are not meant to be merged. As the tool would cache all commits which are not present in all branches, this would cause the cache to become huge. To circumvent this, the number of commits stored in the cache can be limited. This can lead to a situation where a merge scenario should be predicted but the merge base is not in the cache anymore. Thus the features of one or both branches can not be calculated entirely. In this case, we take the features of the oldest commit in the cache of this branch to predict the probability of a merge conflict.
In the aforementioned case, the oldest commit in the cache is still younger than the actual merge base. So if we would take the actual merge base, the value for each feature, such as the number of changes, would be higher than the value we use for predicting the probability of a merge conflict. Usually, the probability increases with the values for the features, so the predicted probability will be lower than the actual probability.
The tool keeps track of the amount of item in the cache and will prune it after an update if necessary.

3.5.2. Persisting

The cache as well as the trained classifier have to be stored on disk.
For the classifier, the best option is to use serialization. As suggested by scikit-learn, we use joblib's[22] implementation to persist and load classifiers.
For the cache, there are multiple ways to do that. One is to store data in a text file, for example in the JSON[23] format. Another one is to use serialization as with the classifiers. Also, the complete cache could be put in a relational database, such as PostgreSQL[24] or a nosql database such as MongoDB[25].
In our implementation, we decided to use python's `pickle` module to persist the cache in a serialized form. Implementing and evaluating different ways to manage and persist the cache was not in the scope of this work and could be done in the future.

[22] `https://pythonhosted.org/joblib/index.html`
[23] `http://www.json.org/`
[24] `https://www.postgresql.org/`
[25] `https://www.mongodb.com/`

3.5.3. Operational Scenarios

There are multiple possible scenarios how GITCoP can be used in software engineering processes. In this section we explain those scenarios and discuss what has already been implemented.
GITCoP itself consists of the cache, an intermediate layer that uses the cache, and an interface that manages communication with the system. By changing the interface, GITCoP can be modified to work in a different scenario.

Commandline Tool

Currently, GITCoP is implemented as a commandline tool. That is, it is called by a command, a specific task, like predicting conflicts, is executed and then the process finishes. That also implies that at every call, the cache and classifiers have to be loaded and stored on the disk in case they changed.
The developer has to manually interact with the system to update the cache and get predictions, which is an additional step she has to take and might break the workflow or simply be forgotten.
This also means that every developer has to maintain their own cache and that using the tool is optional, which might not be desirable if a policy has to be enforced in a team.

GIT Hook

To overcome this, the tool could be directly connected to the repository using GIT hooks[26]. If the tool was called every time a developer commits, the cache would be updated automatically and the developer could be warned about conflicts that could probably occur when merging the branch, she is currently working on.
GIT hooks could also be used on a remote repository to prevent a developer from pushing, if the changes are likely to cause a commit. In this case, she would be forced to merge the branches causing problems and by that solving conflicts before they get too big.
As before, the cache and classifiers have to loaded into the RAM every time GITCoP is called.

Server Application

To avoid constant loading and saving of the cache, the system could be run in server mode. In this case, it would run as a process and wait for tasks. In case of a commit or push the client would notify the server to update the cache or start a prediction. In this case, the cache and classifiers would only need to be loaded when the server starts and could be periodically saved to disk.
The server could run on each developers computer or centralized where every client can reach it. It could also just run with the remote repository and send emails to developers whose branches are likely to cause conflicts.

[26] `https://git-scm.com/book/it/v2/Customizing-Git-Git-Hooks`

4. Methodology

In this chapter, we describe our measures to circumvent problems posed by the data and discuss techniques we used to evaluate our results.

4.1. Balancing

As shown in Section 3.1, there are significantly less commit scenarios resulting in a conflict than those without a conflict. If a classifier is trained with such imbalanced data, this will cause it to be biased. A biased classifier will put unknown data preferably in the class which was overrepresented in the training data, thus achieving low recognition rates of the underrepresented class.
We mitigated this problem by using undersampling. That is, we removed samples from the "no-conflict" class to get an equal amount of samples in both classes. [15]

4.2. Validation

To avoid overfitting and finding an optimal parameter setting, we use *k-fold cross-validation*. This means that for each evaluation we split the data randomly into `k` slices of equal size and performed the evaluation `k` times where each slice is the test set once and the remaining ones are used for training. We used 8 folds so in each run 87.5 % of the data were used for training and 12.5 % were used for testing. The average of the results is computed for each measure. [16]
This was partly implemented by ourself as scikit-learn's `cross_val_score()`[27] from the `sklearn.cross_validation`[28] package does not support metrics for each class. For example, it can return the average precision but not the precision for each class. We use the function `KFold()`[29] to create the slices and the functions from `sklearn.metrics`[30] to collect the metrics for each run but we implemented the calculation of the average ourself.
In Listing 4.1 we show the relevant parts of our evaluation methods, irrelevant parts have been replaced by `# ...`. In line 5 we split up the data for the cross-validation using `KFold()` and in lines 8 to 11 we do an evaluation run for each fold. In lines 20 to 24 the scores are calculated and in the method starting in line 28 we calculate the average value for each score.

[27] `http://scikit-learn.org/0.17/modules/generated/sklearn.cross_validation.cross_val_score.html`
[28] `http://scikit-learn.org/0.17/modules/classes.html#module-sklearn.cross_validation`
[29] `http://scikit-learn.org/0.17/modules/generated/sklearn.cross_validation.KFold.html`
[30] `http://scikit-learn.org/0.17/modules/classes.html#module-sklearn.metrics`

Listing 4.1: Part of our evaluation framework in Python

```
def evaluate(conf, noconf, clf, k=10, class_labels=(1, -1),
             balance_data=True, shuffle_data=True,
             normalize_data=True, random_state=None):
    #...
    kf = KFold(len(data), n_folds=k, shuffle=shuffle_data,
               random_state=random_state)

    for train_i, test_i in kf:
        results.append(run(data[train_i], labels[train_i],
                           data[test_i], labels[test_i],
                           clf, class_labels))

    avg = calc_avg(results)
    # ...


def run(train_set, train_label, test_set, test_label,
        clf, labels):
    # ...
    precision = sklearn.metrics.precision_score(test_label,
                pred, labels=labels, average=None,
                pos_label=None).tolist()
    # ...
    accuracy = sklearn.metrics.accuracy_score(test_label, pred)
    # ...


def calc_avg(results):
    acc = 0
    # ...
    pre = [0, 0]

    n = len(results)

    for e in results:
        acc += e['acc']
        # ...
        pre[0] += e['pre'][0]
        pre[1] += e['pre'][1]

    acc /= n
    # ...
    pre[0] /= n
    pre[1] /= n
    # ...
```

4.3. Metrics

Next, we define all metrics we use in the following chapters. Where possible, scikit-learn's functions in the `sklearn.metrics` package[31] were used to compute metrics. In the following listing, we define a few basic terms. We inspect a set of items that have been classified, where the classifier returned `positive` if it decided an item belongs to class X and `negative`, it it decided that the item does not belong in this class.

True Positives are items that have been correctly classified as items belonging to class X. From here on the number of true positives of class X will be denoted as $p_t(X)$.

False Positives are items that have been classified as items in class X but actually belong to another class. From here on the number of false positives of class X will be denoted as $p_f(X)$.

True Negatives are items that have been correctly classified as items that do not belong to class X. From here on the number of true negatives of class X will be denoted as $n_t(X)$.

False Negatives are items that have been classified as items that do not belong to class X but actually belong to this class. From here on the number of false negatives of class X will be denoted as $n_f(X)$.

We calculate these numbers for each class. In our case, the classification problem is binary, meaning, only two classes exists. We call them C and N. In this case the following holds:

$$p_t(C) = n_t(N) \tag{4.1}$$
$$p_t(N) = n_t(C) \tag{4.2}$$
$$p_f(C) = n_f(N) \tag{4.3}$$
$$p_f(N) = n_f(C) \tag{4.4}$$

4.3.1. Accuracy

The accuracy A of a classifier is the fraction of correctly classified samples. When X is an arbitrary class, the following holds:

$$A = \frac{p_t(X) + n_t(X)}{p_t(X) + p_f(X) + n_t(X) + n_f(X)} \tag{4.5}$$

This metric is calculated for all samples in the set. The value is in the interval $[0, 1]$ where 0 means that no sample was classified correctly, and 1 means all samples were classified correctly.

[31] `http://scikit-learn.org/0.17/modules/classes.html#module-sklearn.metrics`

4.3.2. Precision

The precision $P(X)$ of a classifier for class X is the fraction of $p_t(X)$ of all items that have been classified as items belonging a class X. Intuitively it is the part of items of class X, that were correctly classified, of all items that were put in this class.

$$P(X) = \frac{p_t(X)}{p_t(X) + p_f(X)} \tag{4.6}$$

The precision can be calculated for each class in the set. The value is in the interval $[0, 1]$ where 0 means that none of the items put in class X actually belongs to this class and 1 means that all items put in class X belong there.
A high precision value for one class does not necessarily mean, that the classifier is accurate. A classifier could, for example, put only one item in a certain class. If this item actually belongs to this class, the precision would be 1 but all other items that also belong to the class have not been found by the classifier.

4.3.3. Recall

The recall $R(X)$ of a classifier for a class X is the fraction of $p_t(X)$ of all items that actually belong to this class. Intuitively, it is the fraction of all items, that belong to X and were found by the classifier.

$$R(X) = \frac{p_t(X)}{p_t(X) + n_f(X)} \tag{4.7}$$

The recall can be calculated for each class in the set. The value is in the interval $[0, 1]$ where 0 means that none of the items which belong to X have been classified correctly and 1 means that all items that belong to class X have been found by the classifier.
A high recall value for one class does not necessarily mean, that the classifier is accurate. In the extreme case, a classifier could put all items in the same class, so the recall for that class would be 1 but the classification would obviously be useless.

4.3.4. F1-Score

The f1-score $F(X)$ is the weighted average of precision and recall of class X, both contribute in equal amounts. It is calculated as follows:

$$F(X) = 2 \times \frac{P(X) \times R(X)}{P(X) + R(X)} \tag{4.8}$$

The f1-score is not defined if $P(X) = R(X) = 0$. In this case, we define it to be 0. It can be calculated for each class in a set. The value is in the interval $[0, 1]$ where the score gets better the higher it is.
For both, precision and recall, we showed cases, where the value can be optimal, but the classifier is highly inaccurate. The f1-score is a metric that combines precision and recall and gives a sensible measure for the accuracy of the classifier for a certain class.

5. Evaluation

The evaluation consists of two parts. First, we evaluate the accuracy of the different machine-learning algorithms and models for predicting merge conflicts. We will describe the important tuning parameters, the experimental setup, the used methodology, and the results.
Second, we evaluate the applicability of GITCoP in real-life scenarios. We concentrate on how the GITCoP cache performs on different git repositories.

5.1. Experimental Setup

The goal of this evaluation is to find out which machine learning algorithm is most precise at predicting merge conflicts.
To achieve this we evaluated different algorithms using different feature sets to get a measure for how precisely they classify merge scenarios into the "confilct" and "no conflict" classes.
To get this measure we use the metrics we defined in Section 4.3.

To avoid training models that are biased towards one class, we used balanced data for training and testing. That is, there are as many samples from the "conflict" class as from the "no conflict" class in the sets. This means the probability to guess the correct class for a sample is 50 %. This is called the "random guess". As a "classifier" without knowledge could achieve this by guessing the classes. Hence, an accuracy score of 0.5 is the lowest any classifier should achieve. For this reason it is called the *baseline*.

5.2. Algorithms and Models

In this section we show how different algorithms perform with different data- and feature sets.
We evaluated the algorithms described in Section 3.4, namely Decision Trees, Random Forests and AdaBoost, different Naive Bayes approaches, Support Vector Machines and Logistic Regression. We tuned them with their respective tuning parameters, the exact values we used can be found in Appendix A.
To get a good picture of the precision of the algorithms and take differences between programming languages into account, we used the two Java datasets and the C dataset, described in Section 3.1.1.
First we give an overview about the results, then we discuss differences between the datasets. Afterward, we show the impact of different feature sets on the accuracy of the classifiers and close with a comparison of the classifiers.

5.2.1. Overview

We evaluated the classifiers for different combinations of dataset, flavour (full and reduced), and feature set (extended, simple, and half). From here on we use the term *configuration* for such a combination and identify a configuration by using a three-tuple of dataset, flavour and feature set. For example, the first configuration we report is "git-java, full, extended" which means that we refer to the full git-java dataset in combination with the extended feature set.

In general, the classifiers performed best with the extended feature set while they were least accurate with the "half" feature set. This means that when having more data available, the learners can actually make use of it to improve their classification accuracy. The usage of ensemble methods improves the performance compared to a single decision tree in most cases, but the percentage differs significantly. SVMs can not reach the accuracy of the decision tree and ensemble classifiers. Specifically, while an SVM using a polynomial or rbf kernels mostly come close to the accuracy of a single decision tree, the accuracy of SVMs with linear and sigmoid kernels is notably lower. The logistic regression classifier's accuracy score is usually between those of the most accurate and most inaccurate SVM. In most cases, the Naive Bayes classifiers achieve the lowest accuracy with a few exceptions, in which the Multinomial and Bernoulli Naive Bayes algorithms perform exceptionally good. The Gaussian Naive Bayes on the other hand is hardly better than the random guess in most cases.
The ensemble methods usually performed best and in almost all cases, the random forest classifier achieved the highest accuracy scores.

In the following paragraphs, we discuss the scores of the classifiers for each configuration and show the scores of all classifiers for each configurations in a separate table. Listed are the accuracy and, for each class, the F1-score as well as precision and recall, which we defined in Section 4.3. The highest value for each metric is highlighted in the tables.

Configuration: Full GIT-Java, Extended Feature Set As the results in Table 5.1 show, the random forest classifier achieved the highest accuracy for this configuration. The Gaussian and Multinomial Naive Bayes were only marginally better than the random guess, the Bernoulli Naive Bayes, however, almost reached the accuracy of a single decision tree and was the most precise classifier in the "no conflict" class.

Configuration: Full GIT-Java, Simple Feature Set As with the configuration "git-java, full, extended", the random forest classifier performed best with the simple feature set. But while the Bernoulli Naive Bayes Classifier performed well with the extended feature set, it is hardly better than the baseline in this case. The accuracy of all classifiers was lower than with the extended feature set, except the Gaussian Naive Bayes which has a low accuracy in both cases. The detailed results can be seen in Table 5.2.

Table 5.1.: Performance values for the configuration git-java, full, extended

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.878	0.884	0.843	0.930	0.871	0.922	0.826
Random Forest	0.907	0.912	0.866	0.963	0.901	0.958	0.851
AdaBoost	0.899	0.903	0.872	0.936	0.895	0.931	0.863
Log.Regression	0.778	0.801	0.728	0.890	0.750	0.859	0.667
SVM (linear)	0.759	0.795	0.692	0.935	0.708	0.899	0.583
SVM (rbf)	0.826	0.836	0.792	0.885	0.815	0.870	0.768
SVM (sigmoid)	0.713	0.757	0.656	0.895	0.649	0.835	0.531
SVM (poly)	0.839	0.847	0.803	0.897	0.829	0.883	0.780
Multinomial NB	0.590	0.624	0.577	0.680	0.550	0.610	0.501
Bernoulli NB	0.867	0.882	0.795	0.989	0.849	0.985	0.746
Gaussian NB	0.586	0.696	0.550	0.950	0.350	0.816	0.223

Table 5.2.: Performance values for the configuration git-java, full, simple

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.766	0.763	0.774	0.753	0.769	0.759	0.780
Random Forest	0.836	0.833	0.845	0.822	0.838	0.827	0.849
AdaBoost	0.814	0.813	0.814	0.812	0.814	0.813	0.815
Log.Regression	0.713	0.741	0.675	0.822	0.678	0.772	0.604
SVM (linear)	0.692	0.740	0.640	0.879	0.621	0.807	0.505
SVM (rbf)	0.750	0.765	0.721	0.813	0.733	0.786	0.686
SVM (sigmoid)	0.680	0.729	0.632	0.861	0.609	0.782	0.499
SVM (poly)	0.756	0.768	0.732	0.807	0.743	0.785	0.705
Multinomial NB	0.588	0.621	0.574	0.677	0.547	0.607	0.498
Bernoulli NB	0.570	0.689	0.539	0.955	0.301	0.804	0.185
Gaussian NB	0.586	0.696	0.550	0.949	0.350	0.814	0.223

Configuration: Full GIT-Java, Half Feature Set The results in Table 5.3 show that the classifiers are even less accurate when the "half" feature set is used compared to the simple feature set. The best accuracy was achieved with by the random forest classifier, as in the two other configurations with the full git-java dataset. The Bernoulli Naive Bayes classifier is the most precise one when classifying scenarios in the "no conflict" class, but the recall for this class is low and it does not perform better than the other Naive Bayes classifiers.

The most accurate classifier for the full git-java dataset is the random forest. It achieves the highest accuracy score with all three feature sets and also the highest F1-scores for each class. The AdaBoost method also achieves high accuracy scores, but in the more difficult cases, the difference of the accuracy scores of the random forest classifier and the AdaBoost classifier increases. Hence, it seems like the random forest can cope better with the more difficult feature sets than the AdaBoost classifier.

Table 5.3.: Performance values for the configuration git-java, full, half

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.723	0.699	0.764	0.645	0.743	0.693	0.801
Random Forest	0.787	0.782	0.799	0.766	0.791	0.775	0.807
AdaBoost	0.758	0.757	0.762	0.752	0.760	0.755	0.765
Log.Regression	0.649	0.687	0.620	0.769	0.601	0.696	0.529
SVM (linear)	0.641	0.693	0.605	0.811	0.567	0.714	0.471
SVM (rbf)	0.729	0.734	0.721	0.747	0.724	0.737	0.712
SVM (sigmoid)	0.643	0.696	0.606	0.816	0.568	0.719	0.470
SVM (poly)	0.717	0.727	0.702	0.753	0.706	0.734	0.681
Multinomial NB	0.616	0.695	0.576	0.878	0.479	0.743	0.354
Bernoulli NB	0.530	0.678	0.516	0.989	0.131	0.866	0.0711
Gaussian NB	0.589	0.692	0.553	0.926	0.379	0.772	0.251

Configuration: Reduced GIT-Java, Extended Feature Set As before, the random forest classifier achieves the highest results with the configuration "git-java, reduced, extended", but its accuracy score is 11.7 percentage points lower than with the full version. The AdaBoost has a similar accuracy score, but the difference between the two classifiers is higher than with the full version. The Bernoulli Naive Bayes classifier, which performed exceptionally well before, is hardly better than a random guess in this case and achieves similar results than the other Naive Bayes classifiers.

Table 5.4.: Performance values for the configuration git-java, reduced, extended

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.709	0.686	0.746	0.636	0.729	0.682	0.783
Random Forest	0.790	0.782	0.813	0.754	0.797	0.770	0.827
AdaBoost	0.763	0.762	0.767	0.757	0.765	0.760	0.770
Log.Regression	0.634	0.669	0.610	0.742	0.589	0.671	0.525
SVM (linear)	0.612	0.676	0.580	0.812	0.514	0.687	0.411
SVM (rbf)	0.682	0.695	0.667	0.726	0.667	0.699	0.638
SVM (sigmoid)	0.599	0.664	0.572	0.791	0.504	0.661	0.407
SVM (poly)	0.700	0.707	0.691	0.724	0.693	0.710	0.677
Multinomial NB	0.553	0.591	0.545	0.646	0.507	0.565	0.461
Bernoulli NB	0.534	0.672	0.519	0.955	0.196	0.715	0.113
Gaussian NB	0.549	0.678	0.527	0.948	0.250	0.743	0.150

Configuration: Reduced GIT-Java, Simple Feature Set As in the last case, when using the simple feature set, the classifiers are less accurate with the reduced version of the git-java dataset than with the full version. Table 5.5 shows the detailed results. The most accurate classifier is the random forest once again.

Table 5.5.: Performance values for the configuration git-java, reduced, simple

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.678	0.669	0.688	0.653	0.686	0.670	0.704
Random Forest	0.765	0.760	0.779	0.741	0.771	0.753	0.789
AdaBoost	0.737	0.736	0.739	0.734	0.738	0.736	0.741
Log.Regression	0.625	0.661	0.603	0.732	0.580	0.659	0.518
SVM (linear)	0.602	0.669	0.572	0.804	0.500	0.672	0.399
SVM (rbf)	0.667	0.685	0.651	0.723	0.648	0.688	0.612
SVM (sigmoid)	0.596	0.662	0.569	0.791	0.498	0.657	0.400
SVM (poly)	0.673	0.690	0.656	0.729	0.654	0.695	0.618
Multinomial NB	0.552	0.588	0.544	0.641	0.507	0.563	0.462
Bernoulli NB	0.534	0.672	0.518	0.955	0.196	0.714	0.113
Gaussian NB	0.550	0.678	0.528	0.948	0.254	0.744	0.153

Configuration: Reduced GIT-Java, Half Feature Set As the results in Table 5.6 show, the classifiers are more inaccurate with the "git-java, reduced, half" configuration than with the simple feature set and also more inaccurate than with the same feature set and the full git-java dataset. The most accurate classifier, random forest as before, achieves an accuracy score just over 70 %, all other classifiers are less accurate than that.

Table 5.6.: Performance values for the configuration git-java, reduced, half

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.642	0.625	0.657	0.597	0.658	0.631	0.687
Random Forest	0.714	0.703	0.732	0.677	0.725	0.699	0.752
AdaBoost	0.699	0.696	0.702	0.690	0.701	0.695	0.707
Log.Regression	0.592	0.634	0.575	0.707	0.539	0.619	0.478
SVM (linear)	0.575	0.638	0.556	0.747	0.487	0.615	0.403
SVM (rbf)	0.657	0.668	0.647	0.690	0.645	0.668	0.624
SVM (sigmoid)	0.584	0.624	0.570	0.693	0.531	0.611	0.475
SVM (poly)	0.637	0.647	0.630	0.665	0.627	0.645	0.610
Multinomial NB	0.564	0.656	0.542	0.832	0.405	0.638	0.297
Bernoulli NB	0.514	0.670	0.507	0.989	0.0738	0.779	0.0388
Gaussian NB	0.546	0.670	0.527	0.920	0.275	0.684	0.173

Generally, the accuracy of the classifiers was lower when the reduced version of the git-java dataset is used than with the full one. The reason for this is that the full version includes a huge amount of samples that are easy to classify. In all three cases, the random forest method is the most precise one. The decision tree and ensemble methods still perform better than the SVMs and Naive Bayes classifiers, but the difference between the achieved results are smaller than with the full git-java dataset.

Configuration: Full JDime, Extended Feature Set The results for the "jdime, full, extended" configuration is shown in Table 5.7. The random forest method yields the highest accuracy, but it is only 0.4 percentage points better than a single decision tree and 0.003 higher than AdaBoost's accuracy score. The decision tree classifier is the most precise one when classifying samples of the "no conflict" class, 99.5 % of the samples, the decision tree classified as "no conflict", actually belonged to this class. It also achieves the highest recall score (0.995) for the "conflict" class, meaning that it found 99.5 % of all samples containing conflicts.
As with the "git-java, full, extended" configuration, the Bernoulli Naive Bayes classifier performed well in this case. Also the Multinomial Naive Bayes classifier achieved a higher result than usual and was the most accurate when it comes to classifying samples in the "conflict" class, where it achieved an accuracy score of 0.893

Table 5.7.: Performance values for the configuration jdime, full, extended

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.910	0.916	0.850	0.995	0.901	0.995	0.824
Random Forest	0.914	0.919	0.868	0.977	0.908	0.974	0.851
AdaBoost	0.911	0.915	0.878	0.954	0.907	0.951	0.868
Log.Regression	0.790	0.763	0.872	0.680	0.811	0.738	0.901
SVM (linear)	0.763	0.720	0.877	0.611	0.794	0.702	0.914
SVM (rbf)	0.855	0.852	0.869	0.837	0.858	0.843	0.874
SVM (sigmoid)	0.772	0.735	0.873	0.635	0.799	0.714	0.909
SVM (poly)	0.868	0.868	0.868	0.868	0.867	0.868	0.867
Multinomial NB	0.762	0.713	0.893	0.594	0.796	0.696	0.929
Bernoulli NB	0.879	0.882	0.862	0.902	0.876	0.897	0.856
Gaussian NB	0.584	0.692	0.549	0.938	0.354	0.789	0.229

Configuration: Full JDime, Simple Feature Set When the simple feature set is used instead of the extended one, the random forest classifier is the most accurate one. It performs a little better than a single decision tree and the AdaBoost method is not as accurate as a single tree. As before, the Multinomial Naive Bayes classifier is quite accurate and it also has the best precision score in the "conflict" class and the best recall score in the "no conflict" class. The results for the "jdime, full, simple" configuration in Table 5.8 show that the accuracy score of those Naive Bayes Classifiers are hardly lower than with the extended feature set.

Configuration: Full JDime, Half Feature Set The configuration "jdime, full, half", for which we shown the results in Table 5.9, is the only one where the decision tree classifier achieves a higher accuracy score than the random forest and is also the most accurate classifier for that configuration.
As before the Multinomial Naive Bayes classifier performed quite well. The Bernoulli one even was almost as accurate as the decision tree and achieved the second highest accuracy score. It also achieved the highest F1-score for the "no conflict" class, meaning that it is the best classifier to classify scenarios in the "no conflict" class.

Table 5.8.: Performance values for the configuration jdime, full, simple

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.877	0.879	0.863	0.895	0.874	0.891	0.858
Random Forest	0.882	0.885	0.861	0.910	0.878	0.904	0.853
AdaBoost	0.871	0.872	0.863	0.882	0.869	0.879	0.860
Log.Regression	0.780	0.747	0.878	0.651	0.805	0.723	0.910
SVM (linear)	0.750	0.695	0.890	0.571	0.788	0.685	0.928
SVM (rbf)	0.829	0.818	0.868	0.775	0.838	0.798	0.882
SVM (sigmoid)	0.766	0.743	0.822	0.680	0.784	0.728	0.851
SVM (poly)	0.824	0.815	0.858	0.777	0.831	0.796	0.870
Multinomial NB	0.762	0.713	0.893	0.593	0.795	0.696	0.929
Bernoulli NB	0.874	0.877	0.854	0.901	0.870	0.895	0.847
Gaussian NB	0.584	0.692	0.549	0.938	0.356	0.788	0.231

Table 5.9.: Performance values for the configuration jdime, full, half

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.832	0.823	0.869	0.782	0.840	0.802	0.882
Random Forest	0.827	0.823	0.843	0.806	0.831	0.814	0.849
AdaBoost	0.792	0.788	0.804	0.773	0.796	0.782	0.811
Log.Regression	0.725	0.656	0.875	0.525	0.770	0.660	0.925
SVM (linear)	0.688	0.588	0.864	0.446	0.749	0.627	0.930
SVM (rbf)	0.806	0.793	0.849	0.744	0.817	0.773	0.868
SVM (sigmoid)	0.685	0.571	0.887	0.422	0.750	0.621	0.947
SVM (poly)	0.790	0.762	0.878	0.674	0.811	0.735	0.905
Multinomial NB	0.728	0.650	0.908	0.507	0.777	0.658	0.949
Bernoulli NB	0.831	0.819	0.878	0.769	0.840	0.794	0.893
Gaussian NB	0.581	0.691	0.548	0.936	0.350	0.780	0.227

The classifiers achieve higher results with the full jdime dataset than with the full git-java dataset. We will discuss the differences between the results for different datasets in Section 5.2.2.
With the full jdime dataset the Bernoulli Naive Bayes achieved high accuracy scores for all feature sets and the Multinomial one was also a lot more accurate than with the git-java dataset. When the extended and simple feature set was used, the random forest was the most accurate classifier as before, but for the "half" feature set, the order of the best classifiers changed.

Configurations: Reduced JDime Dataset With the reduced jdime dataset, the extended feature set achieves the best results. Regardless of the feature set, the random forest classifier is most accurate for the reduced jdime dataset. None of the classifiers is really accurate, only the random forest classifier achieved an accuracy score over 0.7 and that only with the extended feature set. The detailed results are shown in the tables 5.10, 5.11 and 5.12.

Table 5.10.: Performance values for the configuration jdime, reduced, extended

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.660	0.647	0.674	0.626	0.671	0.651	0.697
Random Forest	0.711	0.709	0.714	0.705	0.712	0.708	0.717
AdaBoost	0.691	0.689	0.692	0.687	0.692	0.689	0.694
Log.Regression	0.616	0.600	0.627	0.578	0.629	0.608	0.655
SVM (linear)	0.594	0.543	0.621	0.482	0.635	0.577	0.706
SVM (rbf)	0.650	0.639	0.657	0.623	0.659	0.642	0.676
SVM (sigmoid)	0.595	0.556	0.613	0.511	0.626	0.582	0.678
SVM (poly)	0.659	0.647	0.668	0.628	0.668	0.650	0.688
Multinomial NB	0.609	0.643	0.598	0.708	0.562	0.640	0.517
Bernoulli NB	0.569	0.642	0.552	0.777	0.452	0.627	0.366
Gaussian NB	0.547	0.673	0.526	0.933	0.261	0.711	0.160

The accuracy drops between the full and reduced datasets is notably larger when the jdime dataset is used than with the git-java dataset. While the classifiers were more accurate with the full jdime dataset then with the full git-java dataset, it is the other way around with the reduced versions.

Table 5.11.: Performance values for the configuration jdime, reduced, simple

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.642	0.628	0.658	0.609	0.653	0.635	0.679
Random Forest	0.691	0.688	0.694	0.682	0.693	0.687	0.699
AdaBoost	0.661	0.663	0.660	0.667	0.659	0.664	0.656
Log.Regression	0.615	0.600	0.626	0.579	0.628	0.608	0.653
SVM (linear)	0.595	0.543	0.622	0.482	0.635	0.577	0.707
SVM (rbf)	0.640	0.629	0.648	0.611	0.650	0.633	0.669
SVM (sigmoid)	0.590	0.532	0.620	0.467	0.635	0.573	0.714
SVM (poly)	0.646	0.633	0.656	0.613	0.656	0.636	0.677
Multinomial NB	0.612	0.632	0.608	0.671	0.584	0.632	0.559
Bernoulli NB	0.569	0.642	0.552	0.777	0.452	0.627	0.366
Gaussian NB	0.548	0.673	0.527	0.934	0.261	0.712	0.160

Configuration: Full GIT-C, Extended Feature Set In Table 5.13 we shown the results for the configuration “git-c, full, extended”. It is the one where, the classifiers were most accurate. The most accurate one is the random forest classifier, with an accuract of 94.8 %, the AdaBoost classifier was almost as accurate.
But also the SVM with the polynomial kernel achieved an accuracy score of over 90 %. Also the Bernoulli Naive Bayes classifier was quite accurate and achieved the highest precision score in the “ no conflict” class (99 %) and the highest recall score in the “conflict” class (99.3 %).
The other Naive Bayes classifiers were the least accurate one but also achieved accuracy scores around 70 %.

Table 5.12.: Performance values for the configuration jdime, reduced, half

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.631	0.618	0.643	0.599	0.643	0.624	0.667
Random Forest	0.659	0.656	0.662	0.651	0.660	0.655	0.666
AdaBoost	0.633	0.635	0.633	0.638	0.631	0.635	0.629
Log.Regression	0.603	0.550	0.636	0.487	0.644	0.584	0.721
SVM (linear)	0.563	0.409	0.629	0.303	0.653	0.541	0.822
SVM (rbf)	0.636	0.607	0.659	0.563	0.661	0.619	0.709
SVM (sigmoid)	0.560	0.581	0.581	0.669	0.467	0.636	0.475
SVM (poly)	0.631	0.603	0.652	0.562	0.654	0.616	0.700
Multinomial NB	0.586	0.653	0.570	0.788	0.475	0.662	0.395
Bernoulli NB	0.579	0.639	0.559	0.748	0.493	0.619	0.411
Gaussian NB	0.549	0.676	0.528	0.942	0.258	0.731	0.157

Table 5.13.: Performance values for the configuration git-c, full, extended

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.926	0.925	0.926	0.925	0.926	0.925	0.926
Random Forest	0.948	0.948	0.945	0.950	0.947	0.950	0.945
AdaBoost	0.942	0.942	0.942	0.942	0.942	0.942	0.943
Log.Regression	0.822	0.835	0.778	0.900	0.807	0.882	0.743
SVM (linear)	0.823	0.836	0.778	0.903	0.807	0.884	0.743
SVM (rbf)	0.897	0.897	0.894	0.901	0.897	0.900	0.893
SVM (sigmoid)	0.787	0.807	0.737	0.890	0.762	0.862	0.682
SVM (poly)	0.909	0.909	0.906	0.912	0.908	0.912	0.905
Multinomial NB	0.700	0.737	0.656	0.842	0.651	0.779	0.559
Bernoulli NB	0.861	0.877	0.785	0.993	0.839	0.990	0.728
Gaussian NB	0.710	0.762	0.647	0.927	0.630	0.870	0.494

Configuration: Full GIT-C, Simple Feature Set With the simple feature set, the random forest classifier was the most precise one, as well. The Bernoulli Naive Bayes classifier did not perform well and is the most inaccurate classifier for this configuration, the other classifiers lost no or only a few percentage points compared to the extended feature set.
We show the results of the evaluation with the "git-c, full, simple" configuration in Table 5.14.

Configuration: Full GIT-C, Half Feature Set The results in Table 5.15 show that, when the half feature set is used, the random forest is the most accurate classifier when the “half” feature set is used, as usual. Even with this feature set, it achieves an accuracy score of over 90 %.
The Bernoulli Naive Bayes classifier is the most precise at classifying “no conflict” class samples, but the recall for this class is low and the classifier is the least accurate in this configuration.

Table 5.14.: Performance values for the configuration git-c, full, simple

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.908	0.905	0.929	0.882	0.910	0.888	0.933
Random Forest	0.931	0.929	0.953	0.907	0.933	0.911	0.955
AdaBoost	0.925	0.923	0.939	0.909	0.926	0.911	0.942
Log.Regression	0.812	0.825	0.770	0.889	0.796	0.869	0.734
SVM (linear)	0.808	0.822	0.765	0.889	0.791	0.867	0.728
SVM (rbf)	0.886	0.886	0.890	0.881	0.887	0.882	0.891
SVM (sigmoid)	0.783	0.804	0.734	0.888	0.757	0.858	0.677
SVM (poly)	0.892	0.892	0.895	0.889	0.892	0.890	0.895
Multinomial NB	0.700	0.737	0.656	0.842	0.650	0.779	0.558
Bernoulli NB	0.611	0.700	0.569	0.911	0.444	0.777	0.311
Gaussian NB	0.713	0.763	0.650	0.922	0.636	0.866	0.503

Table 5.15.: Performance values for the configuration git-c, full, half

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.893	0.889	0.917	0.863	0.896	0.871	0.922
Random Forest	0.911	0.909	0.936	0.883	0.914	0.890	0.939
AdaBoost	0.901	0.899	0.914	0.884	0.902	0.888	0.917
Log.Regression	0.803	0.815	0.770	0.865	0.790	0.846	0.742
SVM (linear)	0.811	0.820	0.780	0.866	0.800	0.849	0.756
SVM (rbf)	0.892	0.889	0.913	0.866	0.895	0.873	0.918
SVM (sigmoid)	0.814	0.822	0.788	0.859	0.805	0.845	0.768
SVM (poly)	0.875	0.870	0.903	0.840	0.879	0.850	0.910
Multinomial NB	0.729	0.776	0.661	0.939	0.657	0.895	0.519
Bernoulli NB	0.581	0.703	0.544	0.993	0.287	0.960	0.169
Gaussian NB	0.684	0.746	0.624	0.928	0.582	0.858	0.441

We achieved the highest accuracy scores with the full git-c dataset. As with the other datasets, the extended feature set yields the best results, but with all three feature sets, the random forest classifier achieved an accuracy of over 90 %.

Configuration: Reduced GIT-C, Extended Feature Set We show the detailed results of the evaluation with the "git-c, reduced, extended" configuration in Table 5.16. As before, the random forest classifier is the most accurate one but its accuracy score is notably lower than that achieved with the full dataset and the "half" feature set.
The Bernoulli Naive Bayes classifier, which performed well for the full git-c dataset with the extended feature set, is the least accurate classifier for this configuration.

Configurations: Reduced GIT-C, Simple and Half Feature Set The random forest classifier is the most accurate, when the simple and "half" feature sets are used, as well, as we shown in Table 5.17. With the simple feature set, AdaBoost is

Table 5.16.: Performance values for the configuration git-c, reduced, extended

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.821	0.811	0.863	0.764	0.831	0.789	0.878
Random Forest	0.856	0.847	0.908	0.794	0.865	0.817	0.919
AdaBoost	0.839	0.834	0.862	0.808	0.844	0.819	0.870
Log.Regression	0.656	0.698	0.623	0.794	0.601	0.716	0.519
SVM (linear)	0.652	0.706	0.611	0.835	0.573	0.740	0.468
SVM (rbf)	0.789	0.757	0.893	0.657	0.814	0.729	0.921
SVM (sigmoid)	0.679	0.703	0.654	0.759	0.651	0.713	0.599
SVM (poly)	0.804	0.797	0.829	0.767	0.811	0.783	0.842
Multinomial NB	0.604	0.674	0.573	0.819	0.495	0.682	0.389
Bernoulli NB	0.535	0.662	0.520	0.911	0.255	0.641	0.159
Gaussian NB	0.619	0.704	0.576	0.907	0.465	0.781	0.332

the second most accurate classifier, but with the other one, the decision tree classifier achieves the second best accuracy score as Table 5.18.

Table 5.17.: Performance values for the configuration git-c, reduced, simple

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.814	0.803	0.853	0.758	0.823	0.782	0.869
Random Forest	0.849	0.838	0.901	0.784	0.858	0.809	0.913
AdaBoost	0.831	0.826	0.851	0.802	0.835	0.813	0.859
Log.Regression	0.653	0.695	0.620	0.791	0.598	0.712	0.515
SVM (linear)	0.647	0.701	0.608	0.830	0.568	0.732	0.464
SVM (rbf)	0.787	0.753	0.896	0.650	0.813	0.726	0.924
SVM (sigmoid)	0.679	0.703	0.654	0.758	0.651	0.713	0.600
SVM (poly)	0.799	0.791	0.824	0.760	0.807	0.777	0.838
Multinomial NB	0.603	0.674	0.572	0.819	0.494	0.682	0.387
Bernoulli NB	0.535	0.662	0.520	0.911	0.255	0.641	0.159
Gaussian NB	0.625	0.704	0.582	0.891	0.489	0.767	0.359

As with the other datasets, the classifiers achieve lower accuracy scores, when the reduced version of the git-c dataset is used then when the full version is used. But the decision tree classifier and the ensemble method still achieve accuracy scores over 0.8.

Over all configurations, the best results were achieved with the git-c dataset. Of the Java datasets, the classifiers were more accurate with the jdime dataset than with the git-java dataset. In almost every case, the random forest classifier achieved the best results, the AdaBoost and decision tree classifiers achieved good results compared to the random forest as well. Some of the other classifiers achieved good results only with certain configurations.

Table 5.18.: Performance values for the configuration git-c, reduced, half

Classifier	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
Tree	0.803	0.788	0.856	0.730	0.817	0.764	0.877
Random Forest	0.817	0.805	0.864	0.753	0.828	0.781	0.882
AdaBoost	0.799	0.793	0.818	0.770	0.805	0.782	0.829
Log.Regression	0.637	0.701	0.596	0.852	0.538	0.741	0.422
SVM (linear)	0.608	0.698	0.568	0.906	0.442	0.767	0.311
SVM (rbf)	0.791	0.766	0.871	0.685	0.811	0.740	0.898
SVM (sigmoid)	0.672	0.690	0.668	0.730	0.631	0.709	0.616
SVM (poly)	0.788	0.758	0.881	0.666	0.811	0.731	0.910
Multinomial NB	0.637	0.705	0.593	0.869	0.526	0.756	0.404
Bernoulli NB	0.515	0.672	0.508	0.993	0.0698	0.832	0.0365
Gaussian NB	0.597	0.695	0.559	0.918	0.406	0.770	0.276

5.2.2. Datasets in Detail

As shown in the last subsection, the performance of various algorithms does not only depend on the feature sets but also on the datasets which are used to evaluate them. In this subsection, we discuss the differences in the datasets in detail.

In Section 3.1.1, we explained that the scenarios in the C-dataset contain substantially more conflicts than those in the Java-datasets. The results show that the classifiers also perform better on those scenarios. There are several possible explanations for this. One reason for this could be the difference between the procedural- and object-oriented paradigm as object-oriented languages make it easier to separate different features and parts of a program.
But the performance differs not only between the two languages but also between the two Java-datasets. The classifiers perform better with the dataset from [1] than with the one collected for this thesis, although they come from almost the same repositories. The reason is probably the technique of collecting and merging the scenarios. While JDime was used for the first dataset, we simply used the basic git tools for the other one. As JDime is a special merge tool for Java, it produces less conflicts while merging but also takes a lot more time. During the creation of the first dataset, the process was aborted at some time, while we analyzed all scenarios for the second dataset.
In summary, the JDime-dataset contains less scenarios and less conflicts due to the tool used to mine it, while the git-dataset contains more scenarios which also contain more conflicts which leads to more "noise" in the dataset.

5.2.3. Feature Sets in Detail

The three feature sets we evaluated fulfill one purpose each. The extended feature set uses as much information as possible to produce the most accurate results. This includes the number of simultaneously changed files, which is an important feature. The simple feature set is meant to contain as much information as possible while being computationally cheaper to gather. It excludes the number of simultaneously

changed files, as to compute this feature, a comparison of the lists of files is needed. The "half" feature set can be used to compute the quality measure described in Section 3.3. It contains only half of the features of the simple dataset, as it completely leaves out the information of one branch.
As the feature sets contain different amounts of information, they lead to different results. In this section, we will compare the results of the best classifiers for each feature set. For this, we use only the full version of each of the three datasets. In the tables 5.19, 5.20, and 5.21 the performance of the best classifier for each feature set is listed. The letters in braces behind the name of the feature set indicate the classifier, which achieved the results. (A) stands for AdaBoost, (R) for Random Forest and (T) for the decision tree classifier.

With the git-java dataset, the difference of the accuracy between the extended and the half dataset is 0.12. This means that the best classifier, random forest in this case, classified 12 % of all samples wrong with the half feature set that were classified correctly with the extended one.

Table 5.19.: Comparison of feature sets for the full git-java dataset

Feature Set (Classifier)	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
extended (R)	0.907	0.912	0.866	0.963	0.901	0.958	0.851
simple (R)	0.836	0.833	0.845	0.822	0.838	0.827	0.849
half (R)	0.787	0.782	0.799	0.766	0.791	0.775	0.807

With the jdime dataset, the best classifier for the extended feature set is the random forest classifier as well, but the one that was most accurate with the half feature set is the decision tree classifier. With this dataset, the best accuracy score for the half feature set is only 8.2 percentage points lower than the best for the extended feature set.

Table 5.20.: Comparison of feature sets for the full jdime dataset

Feature Set (Classifier)	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
extended (R)	0.914	0.919	0.868	0.977	0.908	0.974	0.851
simple (R)	0.882	0.885	0.861	0.910	0.878	0.904	0.853
half(T)	0.832	0.823	0.869	0.782	0.840	0.802	0.882

The difference between the best accuracy scores with the git-c dataset is even lower than with the jdime dataset, it is 0.037. The most accurate classifier for all three feature sets was the random forest classifier.

As expected, the classifiers achieve an improved performance, the more information is contained in the feature set. However, the extent to which performance improves depends also on the dataset that is used.
With all three datasets, the F1 score of both classes reduce by similar amounts when the smaller feature sets are used. This shows that the feature set influences the performance independently from the class.

Table 5.21.: Comparison of feature sets for the full git-c dataset

Feature Set (Classifier)	Acc.	conflict			no conflict		
		F1	Prec.	Rec.	F1	Prec.	Rec.
extended (R)	0.948	0.948	0.945	0.950	0.947	0.950	0.945
simple (R)	0.931	0.929	0.953	0.907	0.933	0.911	0.955
half (R)	0.911	0.909	0.936	0.883	0.914	0.890	0.939

5.2.4. Algorithms in Detail

In almost all combinations of dataset, flavour, and feature set, the random forest classifier was the most accurate one. The only exception is the configuration "jdime, full, half", where the decision tree classifier achieved the highest accuracy score. The best non-ensemble classifier was always the decision tree classifier. In most cases, the accuracy of the SVMs with the polynomial and rbf kernels was similar to that of the decision tree while the accuracy of the other SVMs was notably lower. The logistic regression classifier mostly performed similar to the SVMs while the Gaussian and Multinomial Naive Bayes classifiers were often only slightly better than the random guess. The performance of the Bernoulli Naive Bayes classifier on the other hand was close to that of the decision tree in some cases while in others, it did not perform better than the other Naive Bayes classifiers.

Decision Trees and Ensemble Methods

Table 5.22.: Accuracy of the decision tree classifier, Random Forest and AdaBoost for different configurations

Configuration	Tree	Random Forest	AdaBoost
git-java, full, extended	0.878	0.907	0.899
git-java, full, simple	0.766	0.836	0.814
git-java, full, half	0.723	0.787	0.758
git-java, reduced, extended	0.709	0.790	0.763
git-java, reduced, simple	0.678	0.765	0.737
git-java, reduced, half	0.642	0.714	0.699
jdime, full, extended	0.910	0.914	0.911
jdime, full, simple	0.877	0.882	0.871
jdime, full, half	0.832	0.827	0.792
jdime, reduced, extended	0.660	0.711	0.691
jdime, reduced, simple	0.642	0.691	0.661
jdime, reduced, half	0.631	0.659	0.633
git-c, full, extended	0.926	0.948	0.942
git-c, full, simple	0.908	0.931	0.925
git-c, full, half	0.893	0.911	0.901
git-c, reduced, extended	0.821	0.856	0.839
git-c, reduced, simple	0.814	0.849	0.831
git-c, reduced, half	0.803	0.817	0.799

As mentioned before, the decision tree classifier performed best of all non-ensemble methods. For this reason it was chosen to as the base for the AdaBoost classifier, the Random Forest method is already based on decision trees. Using an ensemble method increases the accuracy compared to using a single tree by a few percent in most cases. But with some configurations, a single decision tree performs better than AdaBoost and once it is even more accurate than the random forest classifier. An example for this is the configuration "jdime, full, half".
In Table 5.22, a comparison of the performance of those classifiers is shown. Those configurations, where the ensemble methods yield the largest performance gain are highlighted in orange, those where they performed worst compared to the decision tree classifier are highlighted in grey.

Support Vector Machines

SVMs did not reach the accuracy of ensemble methods or the decision tree classifier. Still, depending on the kernel, they achieved high accuracy scores in some cases. In Table 5.23 the accuracy values achieved by the SVMs for each configuration is shown, the highest value is highlighted.

Table 5.23.: Accuracy of SVMs with various kernels for different configurations

Configuration	linear	rbf	sigmoid	polynomial
git-java, full, extended	0.759	0.826	0.713	0.839
git-java, full, simple	0.692	0.750	0.680	0.756
git-java, full, half	0.641	0.729	0.643	0.717
git-java, reduced, extended	0.612	0.682	0.599	0.700
git-java, reduced, simple	0.602	0.667	0.596	0.673
git-java, reduced, half	0.575	0.657	0.584	0.637
jdime, full, extended	0.763	0.855	0.772	0.868
jdime, full, simple	0.750	0.829	0.766	0.824
jdime, full, half	0.688	0.806	0.685	0.790
jdime, reduced, extended	0.594	0.650	0.595	0.659
jdime, reduced, simple	0.595	0.640	0.590	0.646
jdime, reduced, half	0.563	0.636	0.560	0.631
git-c, full, extended	0.823	0.897	0.787	0.909
git-c, full, simple	0.808	0.886	0.783	0.892
git-c, full, half	0.811	0.892	0.814	0.875
git-c, reduced, extended	0.652	0.789	0.679	0.804
git-c, reduced, simple	0.647	0.787	0.679	0.799
git-c, reduced, half	0.608	0.791	0.672	0.788

The best results for all datasets and the extended feature set were achieved by the SVM with the polynomial kernel, while for the half feature set, the SVM with the rbf kernel achieved the highest scores. When the simple feature set was used, the polynomial kernel performed better than the rbf kernel, except for the full jdime dataset, but the accuracy scores of the SVMs with both kernels are closer together with the simple feature set than with the extended one. This shows that the rbf

kernel copes better with the feature sets that contain less information than the polynomial kernel. With the full git-c dataset, the SVM with the rbf kernel achieves a higher accuracy score when the half feature set is used than when the simple one is used and with the reduced git-c dataset, the accuracy score for the half feature set is even better than that for the extended one.
All SVMs achieved their best accuracy score with the "git-c, full, extended" configuration and their lowest one with "jdime, reduced, half".

Naive Bayes

The Gaussian Naive Bayes classifier was hardly ever notably more accurate than a random guess. Except for some configurations with the git-c dataset, it could not get an accuracy score over 0.6. The Multinomial Naive Bayes classifier achieved the best accuracy score of all Naive Bayes classifiers ten times, but as the Gaussian Naive Bayes, it never came close to the performance of a decision tree or one of the ensemble methods. The Bernoulli Naive Bayes classifier on the other hand, performed exceptionally well in some cases. It achieved the highest accuracy score of all Naive Bayes classifiers five times. In those five cases, it also came close to the performance of the overall best classifier for those configurations. While it performed well for feature sets and the full jdime dataset, for the full git-java and full git-c datasets, it only performed well with the extended feature set. In those two cases, the classifier lost around 20 to 25 percentage points, when another feature set was used.
In Table 5.24 the accuracy scores of the Naive Bayes classifiers are shown.

Table 5.24.: Accuracy of the Naive Bayes classifiers for different configurations

Configuration	Multinomial NB	Bernoulli NB	Gaussian NB
git-java, full, extended	0.590	0.867	0.586
git-java, full, simple	0.588	0.570	0.586
git-java, full, half	0.616	0.530	0.589
git-java, reduced, extended	0.553	0.534	0.549
git-java, reduced, simple	0.552	0.534	0.550
git-java, reduced, half	0.564	0.514	0.546
jdime, full, extended	0.762	0.879	0.584
jdime, full, simple	0.762	0.874	0.584
jdime, full, half	0.728	0.831	0.581
jdime, reduced, extended	0.609	0.569	0.547
jdime, reduced, simple	0.612	0.569	0.548
jdime, reduced, half	0.586	0.579	0.549
git-c, full, extended	0.700	0.861	0.710
git-c, full, simple	0.700	0.611	0.713
git-c, full, half	0.729	0.581	0.684
git-c, reduced, extended	0.604	0.535	0.619
git-c, reduced, simple	0.603	0.535	0.625
git-c, reduced, half	0.637	0.515	0.597

Logistic Regression

The logistic regression classifier performed similar to the SVMs but never outperformed those with the best score. Only with the configuration "git-c, full, half" its accuracy score was lower than that of all SVMs. It also did not perform exceptionally well or bad in any special case.

5.2.5. Discussion

In this subsection, we discuss the results shown in above and draw some conclusions regarding the choice of algorithm in a real-life scenario.
The results suggest that Decision Trees are best suited for the job of detecting merge conflicts, only outreached by ensemble methods based on decision trees, like the Random Forest Classifier. As discussed in Section 2.3.3 it used multiple trees to reach its decision thus reducing the drawbacks of a single Decision Tree.
In a real-life scenario accuracy might not be the only thing one wants to achieve. For example, if the developer should be warned of possible conflicts, one would want to avoid false alarms to disturb her in her workflow as little as possible, while it is not necessary to find every conflict that actually appears. In this case one would choose a method that has a high precision score in the "conflict" class, which means that if it labels a scenario as "conflict", the possibility that it actually contains one is high.
In a scenario where as many conflicts as possible have to be found, one would choose a method a high recall score in the "conflict" class, meaning that if some scenarios containing conflicts exist in a set of scenarios, they are likely to be found.

5.3. GITCoP

In this section, we show the performance of the GITCoP cache. We evaluated it on a machine containing a two core (four thread) Intel i5-2400 CPU with a maximal clock speed of 3.10 GHz and 8 GB of RAM. It is running a Gentoo Linux system, Kernel 4.4.6, and has Python 3.4.3 installed.
As a subject system, we used the GIT Repository of CPAchecker[32] cloned on 8th of September 2016 with `HEAD` pointing to `2517fbe`. At this point it had 153 branches with a total of 19 077 commits. For the evaluation, we stored the repository in a ramdisk to avoid I/O operations on the HDD.
We configured GITCoP to consider only files with the suffix `.java` and evaluated it with three different values for the maximum cache size per branch. Those are a number of 50, 100 and 150 commits to store in the cache per branch.

We measured the *memory consumption* of the cache. That is, the memory allocated from the beginning of the initialization until it was finished, the memory consumed by the python environment is not included. We also measured the time consumed to initialize the cache in total, called *wall time*, which also includes the time used by the GIT commands. Additionally, we measured the *processor time* consumed

[32] `https://github.com/sosy-lab/cpachecker`

by the cache, not including that used by the GIT commands. The fourth thing we measured is the total time consumed to extract the merge scenarios from the cache. The number of scenarios extracted is the number of branches in the cache minus one, as we created all scenarios where a branch was merged with `trunk`.
We did not measure the time it takes to predict the probability of a conflict, as this is dependent on the classifier and feature set.

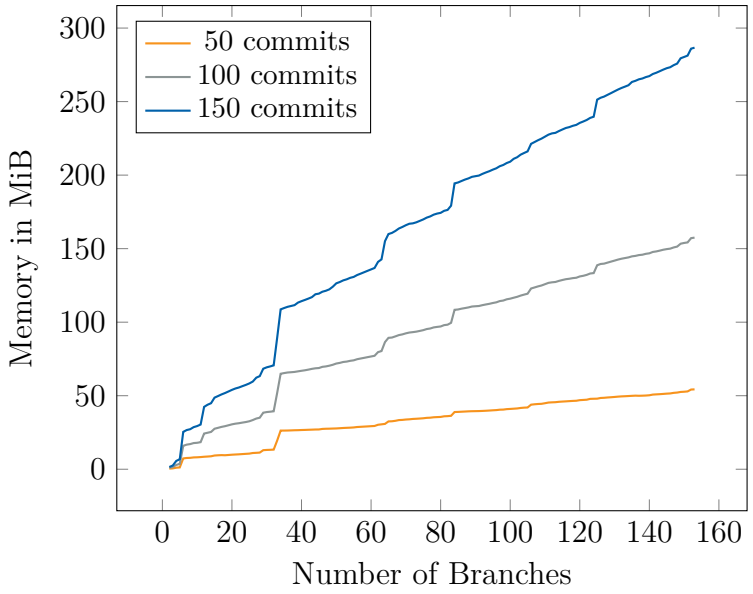

Figure 5.1.: Memory consumed by the cache depending on the number of branches.

Figure 5.1 shows the memory consumption of the cache. The steps in the graph probably come from the way python allocates memory but apart from that, the consumption increases approximately linearly with the number of branches in the cache.
One could assume that when twice the amount of commits is stored for a fixed amount of branches, the memory consumption is also twice as high. But the figure shows that the memory consumption increases by more than a factor of 2. When 50 commits for each of the 153 branches are stored, 54.3 MiB of RAM are used but when 100 commits are stored, the memory usage increases by the factor 2.90 to 157.5 MiB. For 150 commits, it even increases by a factor of 5.28 to 286.7 MiB.
This is caused by the way the implementation stores the list of files edited in the entry for each commit. This list gets longer the farther we go back in history. So the size of an entry in the cache gets larger the more successors it has.

In Figure 5.2, we show the total time it takes to initialize the cache. Here, the slope is not linear but quadratic with the number of branches. The reason is that the cache retrieves and stores the merge base of all pairs of branches. This leads to a total of $\frac{n^2+n}{2}$ combinations, where n is the number of branches. In addition, the youngest common ancestor of all branches is calculated which needs the same amount of computational steps. For each of those steps, an external GIT command is executed, which takes a considerable amount of time. Additionally, we need to analyze every branch, requiring n GIT commands and we need another one per commit to compute the number of changes and the list of change files. So, we need to issue a total of $2 \times \frac{n^2+n}{2} + n \times p + n$ external commands where p is the number

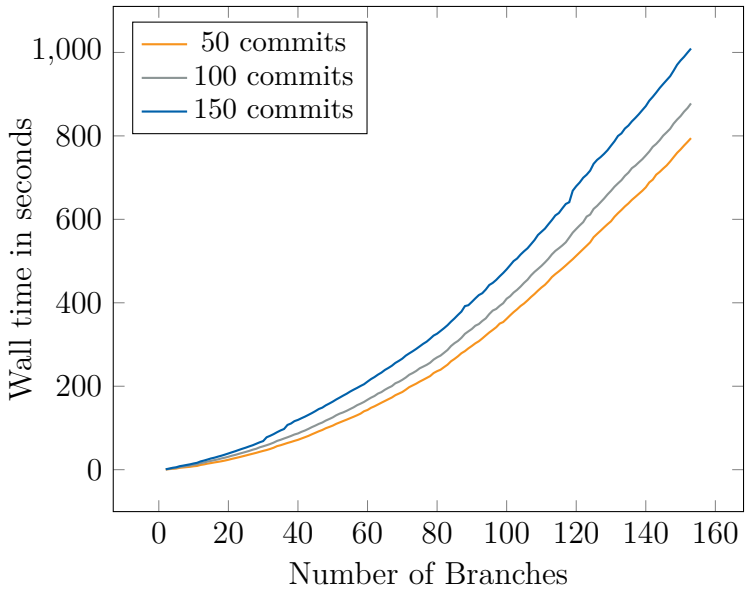

Figure 5.2.: Total time used to initialize the cache.

of commits to be added to the history.
Retrieving the merge bases and computing the youngest common ancestor does not depend on the number of entries we store per branch. Analyzing the branches, on the other hand, does. The more commits we store, the more output of the GIT commands has to be processed and the more entries have to be created. This increases the total time when the number of entries in the cache increase.

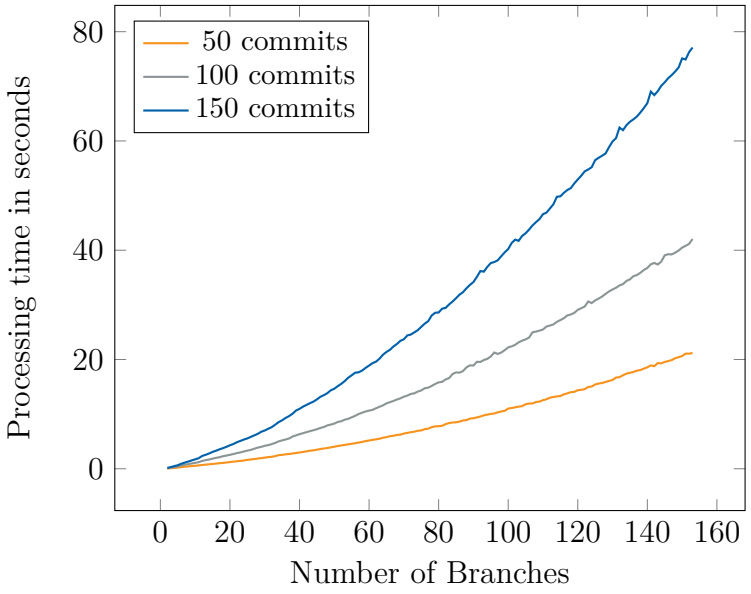

Figure 5.3.: Processing time used to initialize the cache.

Figure 5.3 shows the processing time used to initialize the cache. This is only a small part of the total time, which shows that the external GIT commands take most of the time during the initialization of the cache.
The processing time is mainly consumed by parsing the output of the GIT commands and creating the history entries. The processing time increases by more than the

factor two when twice the number of commits is used, because adding an entry is an update on the cache, which is performed in $\mathcal{O}(e)$ where e is the number of entries that already exist in the cache for the current branch, as discussed in Section 3.5.1. Simply spoken, this means that adding each entry takes longer than adding the one before.

In Figure 5.4 we show the time consumed to retrieve all merge scenarios including the `trunk` branch in milliseconds. The small fluctuations in the graphs can be explained by caching and scheduling effects. Apart from that, the time used to retrieve the scenarios increases linearly with the number of branches. The cache size has only a small impact on the time consumption.

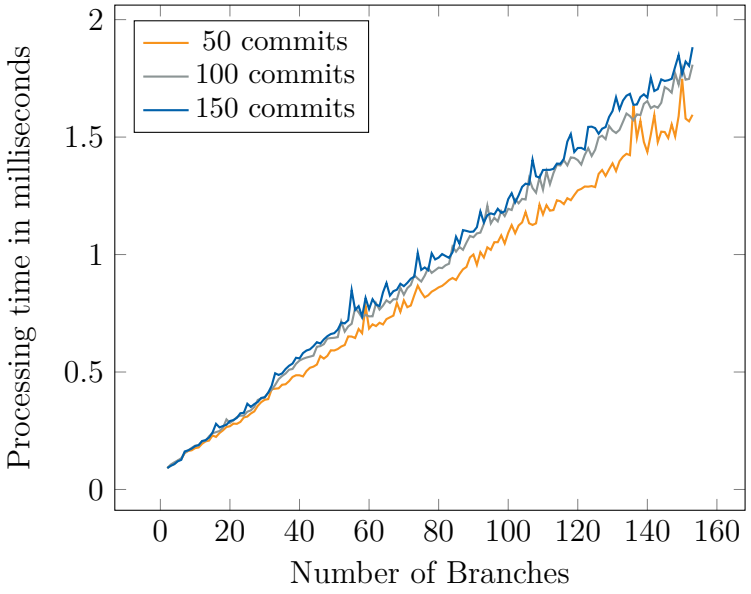

Figure 5.4.: Processing time used to retrieve all merge scenarios.

The cache was implemented in a way to keep time consumed for retrieving data as low as possible, to avoid interfering with the developers workflow. The figure shows that it takes between 1.5 ms and 2 ms to retrieve 152 merge scenarios. This is achieved by only using information that is already present in the cache and without the use of external GIT commands.

To see how the cache performs compared to a normal GIT merge, we measured the time it takes to merge each of the branches into `trunk`. This was done on the same machine as the other evaluations and the repository was stored in a ramdisk as well. The average merge took 0.814 s, the fastest merge was completed in 0.160 s and the most complex one took 2.917 s. In total, 123.758 s were required to merge all 152 branches into `trunk`.
In the box plot in Figure 5.5, we show the time it took to merge each branch. The blue line marks the average time it took to merge a branch into `trunk`, the orange box contains those 50 % of the merges that were closest to the average, while the whiskers mark the branches that took least and most time to merge.

Hence, it is substantially faster to try every merge and find out if conflicts appear than using the cache, as it takes around 16.8 min to initialize the cache for 153

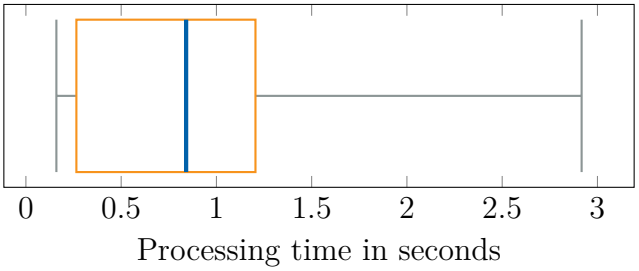

Figure 5.5.: Time needed to merge the branches of CPAChecker into `trunk`.

branches and a cache limit of 150 commits.
Till now, we only used scenarios where we merged one branch into `trunk`, but if we want to see if conflicts appear in any possible combination of the 153 branches, we would have to check a total of 11 781 merges. Presumed that the average merge takes 0.814 s, it would take 9 589.734 s or around 2.66 h to merge all branches and check them for conflicts.
Using the cache, on the other hand would not take notably longer than for 152 merges. The time to initialize the cache will still be around 16 min and the time it takes to retrieve the data and predict conflicts is negligible compared to that.
The cache is designed and meant to be used for a long time, hence the time to initialize the cache is only spent once. After that, it just has to be loaded into the RAM and enables us to predict the probability of merge conflicts in milliseconds. Finding out if conflicts occur by trying to merge branches, on the other hand, will take a similar amount of time, whenever it is done.

The bottom line is that it takes some time to initialize the cache, but we are still talking about minutes and not hours or even days. The cache can consume a notable amount of memory, but is still in a range that can easily be handled by todays customer computers. Retrieving merge scenarios, on the other hand, can be done in milliseconds if the cache is already located in memory.
If only a few merges should be analyzed, it is faster to actually merge and see if conflicts occur instead of using the cache. But if a huge amount of scenarios exist or appear over a period of time, like in software development processes, using the cache gives a huge benefit.

6. Related Work

In this chapter, we present related approaches that deal with predicting merge conflicts or trying to warn developers when the probability they introduce bugs is hight, for example.
Amongst others, we show how Google tries to warn its developers when they are about to change a file that contained bugs in the past and a method that uses continuous merging to warn developers about merge conflicts as soon as they appear. In the end we describe the work of Leßenich et al. which this thesis is based on and show how we used their findings to create the feature sets we introduced in Chapter 3.

Modern software engineering processes are often complicated and sometimes error-prone. In general, fixing errors is the more expensive the later they are discovered. So people try to develop ways to find errors as early as possible or even before they occur. Sometimes, errors occur under similar conditions over and over again. Machine learning can be applied to gain informations about those conditions and predict where errors will occur in the future.

For example, Google describes in their article "Bug Prediction at Google"[33] how they use information from their bug trackers and source code repositories to find "hot spots", files that regularly get changed in bug-fixing commits.
They use the algorithm proposed by Rahman et al. in "BugCache for inspections: hit or miss?" [17] but use a score function that gives more importance to younger commits to filter out files that have been fixed in the past and therefor are not considered "hot spots" anymore. This produces a list of files, which have recently been modified during bug-fixing. The system can be hooked into the VCS and will warn the developer when she is about to change a "hot spot".
We also use information from repositories to find problems as soon as possible and warn the developer about it by hooking a tool into a VCS. But in our case, we do not try to prevent software bugs but merge conflicts.

Guimarães and Silva introduced a system that detects merge conflicts in an early stage and notifies the developer accordingly without overloading her. They use continuous merging of committed and uncommitted changes to compile and test a system in the background. This enables them to find errors, introduced by merges, which are not conflicts, as well. They conducted an empirical study to prove that their solution helps detecting conflicts early without overloading the developer with notifications.
This method works through a plugin in an IDE which connects to a server. Each member of a team has to connect and only work that is done in a connected IDE is tracked. Conflicts and problems are only detected when they occur. [18]

[33] `http://google-engtools.blogspot.de/2011/12/bug-prediction-at-google.html`

To use this method, a constant connection to a server is necessary so the decentralized aspect of GIT is lost. Our method can be used locally without any server. It is not necessary that every developer on the team uses our tool to make it work.

Sarma et al. state that the earlier a conflict is detected, the easier it is to resolve. They detect "workspace isolation" as the reason for merge conflicts to appear. As every developer has his own, isolated, workspace, they are able to make conflicting changes without being aware of it. Their tool, Palantír, is used to raise the "workspace awareness", meaning it makes developers aware of changes that happen in other isolated workspaces, so they realize when multiple developers change the same parts of code, potentially creating a conflict. [19, 20]

Those awareness and continuous merging approaches require a common infrastructure and tool support. Developers have to be connected to a network and use the proper tools for them to work. If a developer is not connected to the network or using other tools, e.g. because of personal preferences, these methods will not work properly. Forcing developers to use certain tools is simple in a company by enforcing company policies, but when it comes to open source projects, this is usually not possible. In those cases, the use of these approaches are limited and inconvenient or even impossible.

Source code repositories and their meta data have been analyzed before to gain insights on software projects and find potential problems.
Tsay et al. analyzed the social structure in open source projects on GitHub to predict the probability for a pull request to be accepted [21].
In large software projects, usually multiple versions are developed and maintained in parallel. When a bugfix is introduced in the current development branch, it has to be pushed to the branches where other versions are maintained, as well. In most projects the branch manager who is responsible for the branch the fix has been committed to has to forward it to the other branches. Tian et al. analyzed the Linux kernel repository to automatically identify bugfix commits that have to be propagated to multiple branches [22].
Potar and Shihab analyzed code comments in open source projects to find "Technical Debt", code that is added but meant to be replaced later [23].

In their paper "Indicators for Merge Conflicts in the Wild: Survey and Empirical Study" Leßenich et al. posed seven hypotheses about what causes merge conflicts which were made based on a survey they conducted. To verify those hypotheses they analyzed over 32 000 merge scenarios by collecting different metrics such as the number of developers working on or the number of modified files in a branch. Then they recreated each of those merges using their custom merge tool, JDime[34], which is a syntax-based, structured merge tool for Java. It takes information from the syntax tree and the language into account to produce better results than the default line-based merge tool provided by git. Of course, this restricted the choice of repositories to analyze to those containing Java code.
In the end, they could not verify a single one of their hypotheses, proving that predicting merge conflicts based on repository meta data is not trivial [1].
For this work, we used some of their approaches and their data to find suitable machine learning algorithms and feature sets to predict merge conflicts. We also use machine learning, where Leßenich et al. used statistical analysis.

[34] `http://fosd.de/JDime/`

In our work we focus on detecting problems by analyzing meta data of repositories without knowing who works on the code right now. We also try to detect branches that probably produce conflicts, possibly before a conflict occurs.
We are building on the metrics suggested by Leßenich et al., but combine them into feature sets. This enables us to create classifiers that predict merge conflicts with an accuracy up to 94.8 %.

7. Conclusion

In this thesis, we propose a technique to predict the probability of merge conflicts based on repository meta-data. To that end, we evaluated different machine learning algorithms and feature sets to create an accurate classifier.
Additionally, we introduce GITCoP, a tool to cache repository data and predict the probability of merge conflicts in a huge number of merge scenarios.

We evaluated four different machine learning algorithms and two ensemble methods, which were tuned using different parameters to create an accurate classifier to predict merge conflicts. In the evaluation, we used a total of eleven classifiers on three different datasets. One was taken from [1], another was mined from a set of around 700 repositories containing mainly Java code and the third one was created from 100 repositories containing mainly C code. We also created three different feature sets containing different measures and serving different purposes.

We found that Random Forest Classifiers are in most cases the most accurate method to classify merge scenarios. Also the evaluation of scenarios in the dataset from C code repositories yielded the most accurate results. The highest accuracy of 94.8 % was achieved by the Random Forest Classifier using the extended feature set with the full GIT-C dataset.
We also found that using a cache can provide means to retrieve information, needed to classify merge scenarios, in microseconds, while analyzing repositories and actually merging a huge number of branches can take minutes or even hours.

In the following sections we provide a summary of the preceding chapters, discuss the challenges we faced and the work that can be done in the future.

7.1. Summary

In Chapter 2, we provided a short introduction to GIT, machine learning in general and the machine learning techniques we used in this work. We also introduce scikit-learn, the machine learning library we used, and GitHub, the site where we retrieved the repositories for our datasets from.

In Chapter 3, we first describe the datasets we are using in this thesis and discuss the features used in the feature set. We show that one feature, the number of simultaneously changed files, is of higher importance as, if this number is zero, conflicts can not occur in the merge.
We also introduce a method to estimate the probability that a branch will create a merge conflict without analyzing any other branch.
We describe the machine learning algorithms we used and show how they can be used in scikit-learn as well as introduce the parameters that can be used for tuning

them. In the end, we discuss GITCoP, our tool to cache repository data and predict merge conflicts.

In Chapter 4, we discuss the techniques and measures we used to evaluate the classifiers.

In Chapter 5, we show that, depending on the feature set and classifier, we achieve an accuracy of up to 94.8 % when predicting merge conflicts in repositories containing C code and an accuracy of up to 91.4 % with repositories containing Java code. The results also show that, in almost every case, the random forest method is the most accurate for the task of predicting merge conflicts.
We also show that after the cache was initialized, creating 152 merge scenarios is a task that takes only around 1.5 ms to 2 ms. Initializing the cache on the other hand can take, depending on the number of commits to store per branch and the number of branches to store in the cache, up to several minutes. We compare the performance of the cache to finding conflicts by doing the merge and show that it is faster not to use the cache if only a few scenarios are relevant and it is a one-time task. If there is a huge number of branches or a repository has to be monitored over time, the cache can give a huge benefit.

7.2. Challenges and Limitations

One challenge to solve was the prediction of the probability for one branch to cause merge conflicts without considering other branches.
When using the datasets, this is easy, as we just have to use the "half" feature set, which only uses information of one branch. But this is not possible in reality. The scenarios in the dataset implicitly contain the information when the branches were split, so the scenario contains the information of n_b commits for both branches. This information is not present when analyzing a branch, so there is no way to determine the number of commits that should be considered. This makes it impossible to just analyze one branch. To overcome this, we introduced the quality measure in Section 3.3.
We also wanted to be able to predict the conflict probability of one branch to avoid analyzing the repository for every possible merge of two branches. As this proved to be impossible, we developed the cache to store the information and gain the ability to retrieve the needed information without analyzing the whole repository every time.

In this work, we expect that changes in different branches are only combined by merging branches. But GIT has another feature to get changes from one branch to another, which is called *rebasing*. As explained in Section 2.1.1, a commit always has another commit as a parent and its changes are based on the state represented by the parents. Where merging combines changes in different branches and leaves the commits untouched, rebasing takes the changes and creates one or more new commits which are applied on top of another commit, making it their parent. The original commits still exist on the original branch and nothing indicates that changes from one branch were applied in another one.
As changes are combined, conflicts can appear in that case as well, but they are fixed in the new commit. So even in the case where conflicts appear, nothing like a

merge commit appears. After a successful rebase, nothing in the GIT history leads to the fact that an interaction between two branches took place.
This has multiple implications on our work. One is that as there are no merge commits, we cannot recreate the rebase as we did it with merges to get data about them and consider them merge scenarios. This is not a serious problem though, as we already found enough data from real merge scenarios without analyzing rebases.
Another implication is that this can lead to "bloated" merge scenarios. If we have two branches and one is rebased onto the other, all the changes are copied. Those are exactly the changes from the original branch. So we have the same changes on the same files in both branches, but we have no way to find out that they are, in fact, identical. This means that, for example the number of changes and the number of simultaneously changed files are high, causing the predicted probability for a merge conflict being too high.
The same effect is caused by *cherry-picking*, a method where changes from a certain commit can be applied to another branch.
The practice of rebasing instead of merging is getting more and more popular these days, as it has some positive implications for the software engineering processes. In projects, where rebasing is used extensively or is even common practice instead of merging, the use of our methods and tools is limited.

7.3. Future Work

In this thesis, we discussed different ways to predict merge conflicts and introduced GITCoP, a tool to cache relevant information to make it possible to predict conflicts quickly, aiming to provide a tool that can be integrated in a software developers workflow to warn her about conflicts.
While we showed that we can create a classifier to predict conflicts in repositories containing mainly Java or C code, the tool is still a prototype that lacks various features. In this section, we show what work can be done in the future on this topic.

In Section 3.5.2, we discuss different ways to persist the information in the cache. In our prototype, we do this by serializing it using Python's default methods for that task. Serialization is probably not the most efficient way to store the cache on disk. So one task in the future is to evaluate different ways or even move the whole cache to a database.

In Section 3.5.3, we propose different scenarios how GITCoP can be used. Currently, only a simple commandline tool is implemented. To be able to use the tool in production, it should be extended so it can be used more flexibly.
Also, the tool works only on Unix-like systems at the moment. Some changes would be necessary to make it work on other platforms as well.

In Section 3.5.1, we discuss in which cases it may be relevant to warn a developer about conflicts. Currently, GITCoP supports predicting merge conflicts for all combinations of branches or all combinations containing one certain branch. Also, all branches that should be in a scenario in the future have to be in the cache completely.
For the future, we propose to make the tool more configurable. For example,

branches starting with “feature-” should always be monitored and all possible combinations should be checked for conflicts, but branches ending with “-legacy” should never be included. Those rules could also reduce the initialization time, as some merge bases could be left out.
Currently, once the cache is initialized, it is not possible to add or remove branches. That is, every time a branch has to be added to the cache, the whole cache has to be reinitialized. As initialization can take a lot of time, it is necessary to add a feature enabling the user to dynamically adding and removing branches.
If one does not want to bother the developer, if no actual conflict occurs, one could alter GITCoP to perform a merge, which is likely to result in a conflict, in the background and only warn the developer, if a conflict actually occurs.

Currently, we have datasets that enable us to train classifiers for the languages Java and C. To enable developers using other languages to use our technique to predict merge conflicts, more datasets have to be mined and classifiers have to be trained and tuned accordingly.

A. Appendix

Parameters

Table A.1.: Performance values for the configuration git-java, full, extended

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	11	30					
AdaBoost	11	30	500				
Random Forest				sqrt			
Log.Regression					140		
SVM (linear)					50		
SVM (rbf)					600	1	
SVM (sigmoid)					500	−2	
SVM (poly)					800	0.1	12

Table A.2.: Performance values for the configuration git-java, full, simple

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	12	60					
AdaBoost	12	60	450				
Random Forest				sqrt			
Log.Regression					140		
SVM (linear)					50		
SVM (rbf)					360	1	
SVM (sigmoid)					500	−2	
SVM (poly)					500	0.1	20

Table A.3.: Performance values for the configuration git-java, full, half

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	10	40					
AdaBoost	10	40	250				
Random Forest				log2			
Log.Regression					80		
SVM (linear)					1		
SVM (rbf)					400	3	
SVM (sigmoid)					450	−2	
SVM (poly)					950	0.1	20

Table A.4.: Performance values for the configuration git-java, reduced, extended

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	10	30					
AdaBoost	10	30	450				
Random Forest				log2			
Log.Regression					110		
SVM (linear)					50		
SVM (rbf)					600	1	
SVM (sigmoid)					500	−2	
SVM (poly)					500	0.1	24

Table A.5.: Performance values for the configuration git-java, reduced, simple

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	12	45					
AdaBoost	12	45	500				
Random Forest				log2			
Log.Regression					140		
SVM (linear)					50		
SVM (rbf)					540	1	
SVM (sigmoid)					500	−2	
SVM (poly)					800	0.1	20

Table A.6.: Performance values for the configuration git-java, reduced, half

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	25	70					
AdaBoost	25	70	350				
Random Forest				sqrt			
Log.Regression					40		
SVM (linear)					1		
SVM (rbf)					300	3	
SVM (sigmoid)					50	−1	
SVM (poly)					800	0.1	20

Table A.7.: Performance values for the configuration jdime, full, extended

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	19	90					
AdaBoost	19	90	350				
Random Forest				all			
Log.Regression					130		
SVM (linear)					50		
SVM (rbf)					540	0.1	
SVM (sigmoid)					100	−1	
SVM (poly)					650	0.1	4

Table A.8.: Performance values for the configuration jdime, full, simple

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	16	65					
AdaBoost	16	65	500				
Random Forest				sqrt			
Log.Regression					80		
SVM (linear)					100		
SVM (rbf)					540	0.1	
SVM (sigmoid)					150	−1	
SVM (poly)					500	0.01	8

Table A.9.: Performance values for the configuration jdime, full, half

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	4	30					
AdaBoost	4	30	450				
Random Forest				log2			
Log.Regression					450		
SVM (linear)					100		
SVM (rbf)					51	3	
SVM (sigmoid)					450	−2	
SVM (poly)					200	0.1	20

Table A.10.: Performance values for the configuration jdime, reduced, extended

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	22	40					
AdaBoost	22	40	300				
Random Forest				log2			
Log.Regression					100		
SVM (linear)					50		
SVM (rbf)					540	0.1	
SVM (sigmoid)					50	−1	
SVM (poly)					650	0.1	8

Table A.11.: Performance values for the configuration jdime, reduced, simple

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	26	115					
AdaBoost	26	115	350				
Random Forest				log2			
Log.Regression					140		
SVM (linear)					50		
SVM (rbf)					540	0.1	
SVM (sigmoid)					400	−2	
SVM (poly)					500	0.05	8

Table A.12.: Performance values for the configuration jdime, reduced, half

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	42	135					
AdaBoost	42	135	150				
Random Forest				sqrt			
Log.Regression					140		
SVM (linear)					1		
SVM (rbf)					151	2	
SVM (sigmoid)					1	−2	
SVM (poly)					800	0.1	20

Table A.13.: Performance values for the configuration git-c, full, extended

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	16	30					
AdaBoost	16	30	350				
Random Forest				log2			
Log.Regression					130		
SVM (linear)					100		
SVM (rbf)					540	0.1	
SVM (sigmoid)					500	−2	
SVM (poly)					800	0.1	16

Table A.14.: Performance values for the configuration git-c, full, simple

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	20	30					
AdaBoost	20	30	300				
Random Forest				sqrt			
Log.Regression					140		
SVM (linear)					150		
SVM (rbf)					540	0.1	
SVM (sigmoid)					500	−2	
SVM (poly)					950	0.1	12

Table A.15.: Performance values for the configuration git-c, full, half

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	50	30					
AdaBoost	50	30	250				
Random Forest				sqrt			
Log.Regression					120		
SVM (linear)					1		
SVM (rbf)					101	3	
SVM (sigmoid)					500	−2	
SVM (poly)					350	0.1	20

Table A.16.: Performance values for the configuration git-c, reduced, extended

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	14	30					
AdaBoost	14	30	500				
Random Forest				log2			
Log.Regression					110		
SVM (linear)					100		
SVM (rbf)					540	0.1	
SVM (sigmoid)					150	-1	
SVM (poly)					950	0.1	20

Table A.17.: Performance values for the configuration git-c, reduced, simple

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	28	40					
AdaBoost	28	40	400				
Random Forest				log2			
Log.Regression					80		
SVM (linear)					50		
SVM (rbf)					540	0.1	
SVM (sigmoid)					150	-1	
SVM (poly)					950	0.1	20

Table A.18.: Performance values for the configuration git-c, reduced, half

Classifier	max d.	min l.	n-est.	max f.	C	$\log_{10}(\gamma)$	deg.
Tree	12	30					
AdaBoost	12	30	50				
Random Forest				log2			
Log.Regression					120		
SVM (linear)					50		
SVM (rbf)					400	2	
SVM (sigmoid)					150	2	
SVM (poly)					950	0.1	20

Bibliography

[1] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen, "Indicators for merge conflicts in the wild: Survey and empirical study." submitted to Empirical Software Engineering Journal, 2016.

[2] P. Domingos, "A few useful things to know about machine learning," *Commun. ACM*, vol. 55, pp. 78–87, Oct. 2012.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[4] P. Harrington, *Machine Learning in Action*. Manning Publications Co., 2012.

[5] C. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[6] H. Zhang, "The optimality of naive bayes," in *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, Miami Beach, Florida, USA* (V. Barr and Z. Markov, eds.), pp. 562–567, AAAI Press, 2004.

[7] D. W. Opitz and R. Maclin, "Popular ensemble methods: An empirical study," *J. Artif. Intell. Res. (JAIR)*, vol. 11, pp. 169–198, 1999.

[8] R. Polikar, "Ensemble based systems in decision making," *IEEE Circuits and Systems Magazine*, vol. 6, pp. 21–45, Third 2006.

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

[10] R. E. Schapire, "The strength of weak learnability," *Machine Learning*, vol. 5, pp. 197–227, 1990.

[11] L. Breiman, "Arcing classifier," *Ann. Statist.*, vol. 26, pp. 801–849, 06 1998.

[12] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[13] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," in *Computational Learning Theory, Second European Conference, EuroCOLT '95, Barcelona, Spain, March 13-15, 1995, Proceedings* (P. M. B. Vitányi, ed.), vol. 904 of *Lecture Notes in Computer Science*, pp. 23–37, Springer, 1995.

[14] scikit-learn developers, "Documentation of scikit-learn 0.17." `http://scikit-learn.org/0.17/documentation.html`, 27. 8. 2016.

[15] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, 2009.

[16] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pp. 1137–1145, Morgan Kaufmann, 1995.

[17] F. Rahman, D. Posnett, A. Hindle, E. T. Barr, and P. T. Devanbu, "Bugcache for inspections: hit or miss?," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011* (T. Gyimóthy and A. Zeller, eds.), pp. 322–331, ACM, 2011.

[18] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in Glinz *et al.* [24], pp. 342–352.

[19] A. Sarma, Z. Noroozi, and A. van der Hoek, "Palantír: Raising awareness among configuration management workspaces," in *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA* (L. A. Clarke, L. Dillon, and W. F. Tichy, eds.), pp. 444–454, IEEE Computer Society, 2003.

[20] A. Sarma, D. F. Redmiles, and A. van der Hoek, "Palantír: Early detection of development conflicts arising from parallel code changes," *IEEE Trans. Software Eng.*, vol. 38, no. 4, pp. 889–908, 2012.

[21] J. Tsay, L. Dabbish, and J. D. Herbsleb, "Influence of social and technical factors for evaluating contribution in github," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014* (P. Jalote, L. C. Briand, and A. van der Hoek, eds.), pp. 356–366, ACM, 2014.

[22] Y. Tian, J. L. Lawall, and D. Lo, "Identifying linux bug fixing patches," in Glinz *et al.* [24], pp. 386–396.

[23] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pp. 91–100, IEEE Computer Society, 2014.

[24] M. Glinz, G. C. Murphy, and M. Pezzè, eds., *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, IEEE Computer Society, 2012.

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und nicht veröffentlicht.

Passau, den 20.01.2017

Thomas Ziegler