DEGREE PROJECT, IN COMPUTER SCIENCE , SECOND LEVEL
STOCKHOLM, *SWEDEN 2015*

Using Git Commit History for Change Prediction

AN EMPIRICAL STUDY ON THE PREDICTIVE POTENTIAL OF FILE-LEVEL LOGICAL COUPLING

ANDERS HAGWARD

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION (CSC)

KTH Computer Science
and Communication

Using Git Commit History for Change Prediction

Användning av Gits versionshistorik för att förutsäga förändringar

An empirical study on the predictive potential of file-level logical coupling
En empirisk studie av den prediktiva potentialen av logisk koppling på filnivå

ANDERS HAGWARD

Master's Thesis in Computer Science at CSC
Civilingenjör Datateknik, internationell inriktning japanska
Masterprogram i Datalogi

Supervisor: Karl Meinke
Examiner: Johan Håstad

Done at Tokyo Institute of Technology
Supervisor: Takashi Kobayashi

hagward@kth.se August 2015

Abstract

In recent years, a new generation of distributed version control systems have taken the place of the aging centralized ones, with Git arguably being the most popular distributed system today.

We investigate the potential of using Git commit history to predict files that are often changed together. Specifically, we look at the rename tracking heuristic found in Git, and the impact it has on prediction performance. By applying a data mining algorithm to five popular GitHub repositories we extract *logical coupling* – inter-file dependencies not necessarily detectable by static analysis – on which we base our change prediction.

In addition, we examine if certain commits are better suited for change prediction than others; we define a *bug fix commit* as a commit that resolves one or more issues in the associated issue tracking system and compare their prediction performance.

While our findings do not reveal any notable differences in prediction performance when disregarding rename information, they suggest that extracting coupling from, and predicting on, bug fix commits in particular could lead to predictions that are both more accurate and numerous.

Referat

Användning av Gits versionshistorik för att förutäga förändringar

De senaste åren har en ny generation av distribuerade versionshanteringssystem tagit plats där tidigare centraliserade sådana huserat. I spetsen för dessa nya system går ett system vid namn Git.

Vi undersöker potentialen i att nyttja versionshistorik från Git i syftet att förutspå filer som ofta redigeras ihop. I synnerhet synar vi Gits heuristik för att detektera när en fil flyttats eller bytt namn, någonting som torde vara användbart för att bibehålla historiken för en sådan fil, och mäter dess inverkan på prediktionsprestandan. Genom att applicera en datautvinningsalgoritm på fem populära GitHubprojekt extraherar vi *logisk koppling* – beroenden mellan filer som inte nödvändigtvis är detekterbara medelst statisk analys – på vilken vi baserar vår prediktion.

Därtill utreder vi huruvida vissa Gitcommits är bättre lämpade för prediktion än andra; vi definierar en *buggfixcommit* som en commit som löser en eller flera buggar i den tillhörande buggdatabasen, och jämför deras prediktionsprestanda.

Medan våra resultat ej kan påvisa några större prestandamässiga skillnader när flytt- och namnbytesinformationen ignorerades, indikerar de att extrahera koppling från, och prediktera på, enbart bugfixcommits kan leda till förutsägelser som är både mer precisa och mångtaliga.

Acknowledgments

I would like to thank Scandinavia-Japan Sasakawa Foundation for their generous scholarship, allowing me to focus on this project without having to worry too much about the finances.

I also want to thank Prof. Kobayashi Takashi, my supervisor at Titech, and Prof. Karl Meinke, my supervisor at KTH, for their invaluable help in providing tips, feedback and general support.

Contents

Chapter 1

Introduction

Most of today's source code is contained in, and handled by, version control systems, systems with the primary task to keep track of changes that are incrementally made to the code, and to facilitate for simultaneous development across several developers. The increased popularity of using distributed version control combined with a web hosting service, e.g. GitHub, in recent years make for easy access to a significant amount of open source software projects' change histories. This raises the question if having access to a project's change history could be utilized to benefit the project.

In this thesis, we investigate the use of change history for being able to automatically warn the developer when she attempts to commit a set of changes into version control in which she has forgotten to update one or more files. For this, we apply a data mining algorithm to the change histories of five software projects to derive association rules among files, and evaluate the effectiveness of using those rules to predict when files are changed together.

In contrast to existing research, we focus on a more modern version control system and investigate if its modern features have any benefits for file-level change prediction. Moreover, we look into the question: "does filtering out non-bug fix concerned change history entries improve prediction performance"?

The rest of the thesis is organized as follows. Chapter 2 describes in large the concepts that this study is based on and reviews existing related research. Chapter 3 describes the specific algorithms and terminology used in this study. Chapter 4 describes the test setups and evaluation strategies. Chapter 5 lists and visualizes the test results. Chapter 6 and 7 discuss the results and answer the research question.

1.1 Purpose

The purpose of this project is to investigate and evaluate the usefulness of using change history to predict and suggest files that are likely to be changed together. Focus is put on the commonly used version control system Git, to determine if modern version control provides any notable benefits for this purpose.

Additionally, a concrete goal is to create a tool suite for Kobayashi Laboratory, to aid in reading repository data, extracting logical coupling and conducting further research on change prediction.

Chapter 2

Background

In this chapter, an overview of the relevant concepts that are motivating for this study is presented.

2.1 Version Control

In the last decade, much has happened to how software source code is stored and handled in a multi-developer environment. In 2005, the development of a new version control system started, *Git*, a project that has since then been both developed and adopted at a rapid rate and is now arguably the most popular version control system.

Version control can be described as the management of changes to any form of information. It has long played an important part in software development; by allowing developers to keep a history of modifications to the source code, it makes it possible to undo changes that were considered unsatisfactory, and paves the way for much easier coding collaboration.

One of the simplest forms of software version control is to manually save old versions of source files. This requires keeping old versions organized in different folders, and while it may work for small projects, it is tedious and prone to errors. Better is to use a *version control system* (hereby referred to as *VCS*). In a VCS, file version history is automatically saved to a *repository*. This removes the risk of accidentally overwriting or removing an old version, and comes with abilities such as saving a log message with each version, and tracking which developer has made which change. Over the years, a wide variety of such systems have emerged, and while their end purposes are the same, the mechanics for how they achieve this can vary.

2.1.1 Types of Systems

Two general types of systems exist today: *centralized* and *distributed* systems. While the centralized model for a long time has been the most common one, distributed

version control has in recent years gained in popularity [10].

In a centralized system, the repository is stored in one place, and developers commit their updates to this *central* repository. One benefit of this approach is control, as it is easy to follow on what is happening to the project and administrators can decide who are allowed to commit. Among the downsides we find the "all eggs in one basket" issue; as the project history is only saved in one single place, one needs to ensure that it is sufficiently backed up to avoid the unfortunate scenario of a total data loss, should the server fail. Examples of centralized VCSs are CVS, Subversion and Perforce.

A distributed system, on the other hand, builds on the idea that there *is* no central repository, but rather a collection of working copies. Each developer has her own copy of the complete change history and changes are committed locally, in contrast to over a network connection as is often the case with centralized systems. Changes from different developers are then merged across repository copies when appropriate. This makes basic operations such as committing fast as they are done locally, in contrast to over a network connection as is often the case with centralized systems. Common distributed systems today are Git, Mercurial and Bazaar [2].

2.1.2 Issue Tracking

A core part of the software development workflow is to be able to keep track of software defects and similar issues, so that they can be addressed and fixed in a systematic and effective manner. For this purpose, *issue tracking systems* are used. Useful features of such systems are to be able to add descriptive text about issues and to discuss them, to assign a developer to a specific issue and to mark issues as "resolved" when they have been dealt with. While an issue often represents a bug, it could also represent other instances of potential enhancements, such as feature requests and regular questions. These kind of systems may be either internal, and visible only for the software development team, or public, in which case the users themselves directly may submit reports and be involved in the development.

Issue tracking systems are usually integrated in web-based hosting services, and may also be connected to the VCS to automatically mark an issue as resolved when detecting certain keywords in a commit message. For example, including the text snippet "resolved #123" in the message indicates that issue number 123 is fixed and, hence, the issue would automatically be marked as closed in the issue tracking system. This functionality could potentially make certain aspects of commit analysis easier as it, to some degree, enforces a notion for when and which issues are closed.

2.1.3 Web Hosting Services

The popularity of Git is also reflected in the web-based source code hosting services, such as GitHub and BitBucket. GitHub has reportedly 9.1 million registered users and 21.6 million repositories [22], in contrast to SourceForge, one of the largest code

hosting services some years ago, which has 3.7 million users and 430,000 repositories [17].

A web hosting service such as GitHub provides a central hub where developers can communicate and collaborate, and includes one or many types of VCSs and an issue tracking system.

2.2 Machine Learning and Data Mining

Machine learning is a scientific discipline concerned with the study of algorithms for learning from data. It stems from research in artificial intelligence and was defined in 1959 by Arthur Samuel as a "Field of study that gives computers the ability to learn without being explicitly programmed" [9].

In recent years *data mining*, a discipline similar to machine learning, has seen a rise in popularity. Since the amount of data in the world is growing at an exponential rate [20], algorithms and techniques for analyzing such vast amounts of data are needed. Although sharing considerable overlap, data mining and machine learning are not fully interchangeable terms; data mining can be described as the process of analyzing, typically large, amounts of data in order to discover previously unknown patterns and relationships, where, in the end, the goal is to gain new knowledge about the data. Often times, the data is not stored with data mining being the primary purpose, and consequently its structure is not laid out for that kind of analysis. This means that data mining can be applied to various kinds of data [8].

With respect to the input data, machine learning can be categorized into three broad categories [13]:

- *Supervised learning:* The learner learns from labeled training data, i.e., data with a well specified structure. An example is spam filtering; the training and test data consists of a collection of emails beforehand marked as either *spam* or *not spam*.
- *Unsupervised learning:* The data is unlabeled and it is up to the learner to find structure in the provided input. Because of this, it is difficult to evaluate the performance of an unsupervised learner. It can be used to discover previously unknown patterns in data.
- *Reinforcement learning:* The learner learns from being rewarded or punished; its task is to decide on a series of actions that maximizes the accumulated reward.

With respect to the output data, machine learning is typically divided into a group of six categories [5]:

- *Deviation detection:* The learner's task is to identify items which do not conform to an expected pattern. Examples of it's applications are detecting network intrusions and bank frauds [3].

- *Clustering:* The learner's task is to identify clusters of items that are in some way similar.
- *Classification:* Training data is divided into a set of classes and the learner's task is to learn to assign items to one or more classes. Once again, an example of this is a spam filter, where items can be classified as either *spam* or *not spam*.
- *Regression:* The task of estimating the relationships among variables. It can be used to forecast the result of various processes, such as the weather, sports results etc.
- *Summarization:* The task of finding a compact description of a set of data.
- *Dependency modeling:* The task of finding and representing dependencies among variables.

In our case, we are using dependency modeling, more commonly referred to as *association rule learning*, in combination with supervised learning. This method is described in more detail in Chapter 3.

2.3 Related Work

We are not the first to incorporate software history data in evolution research. It has been used to find "hidden", or logical, coupling among modules by noting which tend to be changed together [6], to predict when the developer has forgotten to change a file [15] or a fine-grained artifact [16], and to guide a newcomer in becoming familiar with a software system [4].

2.3.1 Architectural Analysis

The idea of mining development histories was first introduced by H. Gall et al. In [6] they investigated module-level change patterns in the release history of a large telecom switching system, and verified them against change reports describing bug fixes and the actual files included in them. By extracting change sequences, i.e., sequences describing in which releases a module has been changed, they found that some modules were often changed together, indicating that inter-modular coupling was present. The results from this study are interesting, as they show promise in using development metadata – something that is present in all software projects – to improve software.

In a later study [7], H. Gall et al. focused on a VCS called CVS. Using three complementary analysis techniques, they were able to find both class-level and higher-level logical coupling. They applied their method on a Picture Archiving and Communication System (PACS), consisting of 500 thousand lines of code. Results indicated that their technique was effective at detecting various kinds of architectural design flaws.

2.3.2 Code Suggestion Systems

In [15] A. Ying et al. investigated file-level logical coupling to use for code recommendation during development. Using frequent pattern mining they were able to extract association rules from CVS repositories, and use them to suggest files that likely were needed to be changed to the developer. They evaluated their method on two large software projects: Eclipse and Mozilla. In addition to precision and recall, they introduced a new evaluation criteria, *interestingness*, based on the level of structural dependency between two files; less structural dependency suggested that the coupling is less "obvious" and therefore more interesting to the developer. While the results did not show any spectacularly high precision or recall (see Section 3.2), the interestingness study looked promising. The core problem and result of this paper coincides with our project as we too are concerned with coupling among files. However, as this study focused on CVS, code refactorings such as renaming or moving files could potentially have rendered some patterns obsolete, as CVS does not retain the file's change history in the case of such events.

In [16] T. Zimmermann et al. also studied code recommendation based on change patterns, and created an Eclipse plugin, ROSE, capable of suggesting relevant artifacts for change. In contrast to [15], their method found patterns on both fine and coarse levels of granularity; by using a parser supporting a handful of programming languages they were able to find patterns in programming-level entities such as methods and fields. They evaluated ROSE on eight large open source projects, and found that a few weeks training data were generally sufficient for ROSE to start returning useful suggestions. The method seemed to work best for projects undergoing maintenance rather than having new features added to them. They also found that adding linear commit weighting, to account for possible inaccuracies in older data, were beneficial to projects that were often refactored.

In [4] D. Čubranić et al. discussed the implementation and evaluation of their artifact recommendation tool named Hipikat, with the aim of helping newcomers to large software projects become familiar with the code. It was showed that Hipikat was able to retrieve information from issue tracking systems, source code repositories, developer forums and other project documents, and identified links between artifacts from these sources. They thereafter conducted two studies of the system's recommendation quality, where they both evaluated recommendations on already completed change tasks in the development history, as well as on real test subjects to probe its capability of helping newcomers. In the first study, results showed high recall and low precision. The second study revealed that test subjects tended to use Hipikat mostly in the beginning of a change task. They concluded that the usefulness of Hipikat is dependent on the amount of information in the artifacts, and especially logged communication.

2.3.3 Software Evolution Research

In [12] R. Robbes et al. evaluated three mainstream VCSs, as well as one niche system, with regards to their suitableness for use in evolution research. The niche system were able to track changes on a fine level of granularity, but only supported one specific programming language. They compared the systems with respect three broad properties: availability, technical features and usability. The fine-level tracking capability of the niche system was deemed to have the potential to be of great value for evolution research, if more programming languages were to be supported. From the results it was concluded that CVS had the best availability, due to being the most widespread of the four. Subversion placed first in technical features, as it had features such as changesets, history on directory level, tracking of renamed or moved entities and atomic commits. The authors identified a set of shortcomings of the current VCSs, namely that they did not support tracking of programming language entities such as methods and fields, and the fact that they were snapshot-based, meaning that the information in between commits is lost.

In [14] Z. Xing et al. studied co-evolution among classes. They obtained code differences between different software versions and, by mining assocation rules, investigated which classes were frequently changed together, i.e., which classes were co-evolving. By combining the two techniques, they could inspect co-evolving classes in detail. They then listed three potential applications of their method: change prediction, detection of parts that would benefit from refactoring and system instability recognition. In the case study, carried out on a medium-sized open source project, several instances of co-evolution were discovered, indicating that some parts suffered from too high coupling.

In [11] S. Negara et al. studied the usefulness of file-based VCS histories for software evolution research. Using their own Eclipse plugin, CodingTracker, they recorded fine-grained development data and compared it to the information stored in the VCS. They discovered that 37% of the changes did not show up in the VCS history, a phenomenon referred to as "shadowing": a change that erases another change "shadows" that change if the two changes are made in between two commits, as in such case the first change would not be discoverable in the VCS history. They also noticed that commits tended to contain several different types of changes, e.g. a refactoring commit sometimes also contains feature additions.

In [1] C. Brindescu et al. compared commits made in centralized (CVCS) and distributed VCSs (DVCS) to determine their impact on software changes. Among their findings were that commits made in DVCSs were 32% smaller, that developers committing to DVCSs were more prone to split their commits into smaller logical units of change, and that commits in DVCSs were more likely to contain references to issue tracking labels.

Chapter 3

Theory

In this chapter the algorithms and technologies used in this project are explained.

3.1 Association Rule Learning

Association rule learning is a method used for discovering previously unknown relations between variables. A typical example of the application of association rules is that of finding shopping patterns in the transactions database for a grocery store. By applying an association rule algorithm to this data, it may be possible to find that certain products are more likely to be bought together than others. One pattern, for example, could be that customers who buy butter are likely to also buy milk.

An association rule can be divided into two parts: a structural and a quantitative part. The structural part consists of a left-hand side, the *antecedent*, and a right-hand side, the *consequent*, and tells which variables are involved in the relationship. Items in the antecedent then *implies* the items in the consequent, to a certain likelihood that is determined by the quantitative part. Several quantitative measures exist, whereof the two most common are:

- *Support:* A rational number between zero and one, determining the fraction of the total transactions that the rule has been derived from. The support of a rule $X \Rightarrow Y$ could be written as $Support(X \cup Y)$ and denotes how large part of the transactions contain both the antecedent X and the consequent Y.

- *Confidence:* Also a rational number between zero and one. The confidence of a rule $X \Rightarrow Y$ is defined as $Confidence(X \Rightarrow Y) = Support(X \cup Y)/Support(X)$. Intuitively it can be thought of as the probability that consequent Y is found given that antecedent X is present, or $P(Y|X)$.

The result from running an association rule algorithm on the grocery database mentioned in the example above might look like:

Left-hand side	Right-hand side	Support	Confidence
butter	bread, milk	0.5	0.5
beer	snacks	0.25	0.75
minced meat, tomatoes	spaghetti	0.1	0.8

This would mean that in 50% of the transactions, people bought butter, bread and milk together, as indicated by the support of first rule. The confidence of the second rule says that in 75% of the transactions where beer was bought, snacks were also bought.

3.1.1 Frequent Itemsets

Association rule learning can be broken down into two steps. The first step is to find the frequent itemsets, i.e., sets of items that occur frequently together. Determining the number of times two items at least have to occur together to be included in a frequent itemset is based on the minimum support value, e.g. a support of 0.5 means that items have to occur together in half of the transactions to be frequent.

Next step is to compute the association rules. From the frequent itemsets, rules can be extracted with the following procedure:

1. For each frequent itemset L, generate all (non-empty) subsets S of L.
2. For each subset s in S, create the rule $s \Rightarrow (L - s)$ and check if $SupportCount(L)/SupportCount(s) \geq minConfidence$, where $SupportCount(X)$ is the number of transactions that contain itemset X. If that holds, the rule can be kept, otherwise it is discarded.

3.1.2 The Apriori Algorithm

In this thesis, we use the Apriori algorithm to find dependencies among files; a well-known and widely used algorithm for mining association rules [8]. It finds frequent k-itemsets (itemsets with k items) by finding frequent itemsets with only one element, and then iteratively generating frequent itemsets of a bigger magnitude until no larger frequent itemsets can be found.

In the Apriori algorithm, a *candidate* k-itemset is a set of k-combinations of items. It is used to list the frequent itemset candidates. The ones that pass the frequency requirement go into a *large* k-itemset.

To do this, for each level it generates a so called candidate set, which contains all combinations of itemsets found in the large itemset, and counts in how many transaction each itemset is present. This step is also amenable for optimization. The downward closure lemma states that if an itemset is frequent, then all of its subsets must be equally, or more, frequent. Using this property, itemsets can be pruned if they contain a subset that has already been eliminated in an earlier round. Finally, the number of occurrences of the itemsets left in the candidate set are counted by checking against the transactions, and we add the ones that achieve the minimum

#	Itemsets
1	B, C, E
2	A, F
3	B, C, D, E
4	B, A, E, F
5	B, C

Table 3.1. A small example database of five transactions, with six unique items.

Itemsets	Support
A	0.4
B	0.8
C	0.6
D	0.2
E	0.6
F	0.4

Itemsets	Support
A	0.4
B	0.8
C	0.6
E	0.6
F	0.4

Figure 3.1. Shows the first candidate itemset C_1 and large itemset L_1 from applying the Apriori algorithm to the database in Table 3.1 with minimum support 0.4.

support count to the large itemset. When the generated large itemset is empty the algorithm stops.

An Example

Consider the small transactional database in Table 3.1. It consists of five transactions involving six different items. We want to apply the Apriori algorithm to find frequent itemsets among the items. For this example we use a minimum support of 0.4.

The first step is to find the first candidate set, C_1, which is basically the set of all the items. Then the database is scanned to count the number of occurrences of each item, and the ones that meet the support requirement goes into the first large itemset L_1 (see Table 3.1). Item D only appears in one transaction and is not carried over to L_1. Checking for larger itemsets containing D can from now on be completely pruned.

Secondly, C_2 is obtained, consisting of every 2-combination of items from L_1. From this set, only four rows have a support larger or equal than 0.4 and go into L_2 (see Table 3.2). In the same manner as in the previous iteration, this moment is prone to further pruning. When building the third candidate set only 3-combinations of items that are supersets of any of the itemsets in L_2 need to be considered.

From C_3, only one itemset meets the support requirement. As the next and final candidate itemset, C_4, does not contain any rows with support equal to or larger

Itemsets	Support
A, B	0.2
A, E	0.2
A, F	0.4
B, C	0.6
B, E	0.6
B, F	0.2
C, E	0.4
E, F	0.2

Itemsets	Support
A, F	0.4
B, C	0.6
B, E	0.6
C, E	0.4

Table 3.2. Shows the second candidate itemset C_2 and large itemset L_2 from applying the Apriori algorithm to the database in Table 3.1 with minimum support 0.4.

Itemsets	Support
A, B, F	0.2
A, E, F	0.2
B, C, E	0.4
B, E, F	0.2

Itemsets	Support
B, C, E	0.4

Table 3.3. Shows the third candidate itemset C_3 and large itemset L_3 from applying the Apriori algorithm to the database in Table 3.1 with minimum support 0.4.

than 0.4, the algorithm stops.

3.2 Precision and Recall

In information retrieval, the quality of retrieved information is often measured in terms of precision and recall. In terms of documents, a high precision means that a large fraction of the retrieved documents were actually relevant, while a high recall means that a large fraction of all the relevant documents were retrieved. Formally we define these two measures as:

$$precision = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|}, \tag{3.1}$$

$$recall = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{relevant\ documents\}|}. \tag{3.2}$$

However, depending on the context it may not always be practical to speak in terms of retrieved documents. In binary classification, and in this thesis, the following definitions are used:

$$precision = \frac{tp}{tp + fp}, \tag{3.3}$$

$$recall = \frac{tp}{tp + fn}, \tag{3.4}$$

where tp, fp and fn stands for *true positives*, *false positives* and *false negatives* respectively. The positive and negative aspects of these terms correspond to the binary classification: it can either be classified as positive or negative. The true and false parts tell if the classification was correct or not. For example, a true positive would be a correctly classified "positive" item, while a false negative would be an incorrectly classified "positive" item.

3.2.1 F-measure

The F-measure is a measure that considers both precision and recall. The general definition is:

$$F_\beta = (1 + \beta^2) * \frac{precision * recall}{\beta^2 * precision + recall}. \tag{3.5}$$

The β parameter acts as a weight to favor either precision or recall. Setting $\beta < 1$ will weight precision higher than recall, and $\beta > 1$ will do the opposite. In our case, we use $\beta = 1$, giving the same weight to precision and recall. This is equal to the *harmonic mean* of the two numbers and is arguably the most common setting, sometimes called the F_1 *score*:

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \tag{3.6}$$

Chapter 4

Methodology

We chose to implement the logical coupling extraction tool in Java, for the reasons that it is a reliable and widely used language. Thanks to this choice, the tool is executable on a variety of different platforms that have Java Runtime Environment installed.

To be able to traverse and read data from Git repositories, Java library JGit [21] was used. For association rule mining we used the statistical computing language R [23] with the data mining library *arules* [18]; we did not implement the data mining algorithm ourselves, but used an already existing and proven implementation of the Apriori algorithm. This way, the time spent on coding could be reduced, together with the risk of using a faulty algorithm implementation.

As we used two different programming languages, we needed a way to connect them. This was done using Rserve [24], a combined R and Java library. It acts as a TCP/IP server and listens for R commands from clients, i.e., software using the Rserve library. When it receives a command, it forwards it to R for evaluation and returns the result.

4.1 Reading Repository Data

The selected repositories were traversed backwards, commit by commit. The reason for traversing backwards instead of forwards is that handling of deleted files becomes a more simple endeavor. For example, if a file is found to have been deleted in the repository it can simply be ignored when found in subsequent (older) commits, as it is to be deleted anyway. This means that we reset change history for a file when it is deleted; if a file is deleted and later re-added, only the change history from its re-addition is considered.

We call the addition, deletion or modification of a file a *change event*. For every commit, each change event was inspected and, depending on its type as well as program settings, different actions were taken. For JGit, a total of five different types of file changes were supported, and they were interpolated into three different change events:

1. *Add/Copy/Modify:* These three change types are what constitutes logical coupling. An addition means that a file previously not existing in the repository has been added. A copy indicates that a file has been copied to another location (duplicated). Finally, arguably the most common type is a modification, which indicates that the contents of a file have been modified.

2. *Delete:* When a file is found to have been deleted, it will be preemptively ignored if found in any older commits. The reason for this is that we assume that, since such a file has been deliberately deleted, it is more likely that it will "stay deleted" than it is that it will be later re-added. Furthermore, even if a deleted file is re-added, there is no guarantee that it will still have its old couplings preserved.

3. When a file is found to have been renamed, its old name is simply linked to the new name. Note that JGit does not make any difference between renaming a file and moving it, as renaming a file can be viewed as moving it to a new location within the same directory.

4.1.1 Storing Changesets

Because one file typically is to show up in many different commits, storing its name each time would be redundant and inefficient. Instead, each file name is stored only once in a list with its own numerical index. To be able to quickly look up the index for a file when encountered in a commit, the list is backed up by a hash table with file names as keys storing the list index of each file name.

This structure is further tailored to allow to detect deleted and renamed files. A `null` element in the file name list indicates that the file at that specific index has been deleted. Because a delete operation can be performed preemptively on an item that is not yet in the list (in the cases where a deleted file is never re-added), a delete operation consists of first adding the file name to both the list and hash table if it does not already exist, and then setting the list element to `null`. Hence, a delete operation does not actually delete an element from the structure, but rather stores information that the file has been deleted. A file can quickly be checked if it has been deleted by looking up its list index, and then checking the corresponding list element for `null`. When a rename event occurs it can simply update the name in the list, and make sure that both the old and the new name in the hash table point to the same index.

With this system in place, changesets can finally be represented as a two-dimensional array of integers, where each row corresponds to a set of files – a commit.

4.2 Extracting Logical Coupling

With the changesets for a Git repository already read and stored as a twodimensional integer array, as described in the previous section, they had to be passed to R for

association rule learning. To do this, the vector first had to be converted to an R vector. It was then sent to Rserve, as a string, together with R commands to be executed. Rserve, in turn, let R execute the commands and retrieved the result.

An association rule was represented as two integer arrays and one double array. The two former arrays contained the file name indices for the left- and right-hand side of the rule respectively, while the latter contained the quality measures support and confidence. By consulting the two-way map structure described in the previous section, the file names could be re-obtained from the indices and rules could be presented in a human readable form.

4.3 Predicting Changes

For each commit, the left-hand side of each association rule was checked against the commit changeset to see if it was applicable. We defined a rule as applicable for a changeset if its left-hand side was a subset of the changeset. E.g., for a changeset $\{0, 1, 2, 3\}$, the rule $\{0, 2, 3\} \Rightarrow \{4\}$ would be applicable while $\{2, 4\} \Rightarrow \{0, 1\}$ would not.

Using this definition, we further defined two sets: the *predicted set*, S_p, as the set of all predicted files for a changeset S_{cs} (see Eq. 4.1); and the *cover set*, S_{cov}, as the set of files in the changeset that were *covered*, i.e., either correctly predicted or used to predict (see Eq. 4.2).

$$S_p = \bigcup_{\substack{(R_{lhs}, R_{rhs}) \in rules \\ R_{lhs} \subseteq S_{cs}}} R_{rhs} \tag{4.1}$$

$$S_{cov} = \bigcup_{\substack{(R_{lhs}, R_{rhs}) \in rules \\ R_{lhs} \subseteq S_{cs}}} R_{lhs} \cup (S_{cs} \cap R_{rhs}) \tag{4.2}$$

Where R_{lhs} and R_{rhs} correspond to the left-hand and right-hand side of a rule respectively. In order to measure the performance of the predictions we defined two measures, *precision* and *coverage* (see Eq. 4.3 and 4.4).

$$precision = \frac{|S_{cs} \cap S_p|}{|S_p|} \tag{4.3}$$

$$coverage = \frac{|S_{cs} \cap S_{cov}|}{|S_{cs}|} \tag{4.4}$$

Note that coverage differs from recall in that it includes the antecedents of the rules in the computation. The reason for using coverage instead of recall is because we look at the coupling between two files as an undirected edge between two nodes; e.g. if file A is coupled to file B, then $A \Rightarrow B$ and $B \Rightarrow A$ should both hold.

Consider the example given in Fig. 4.1 and 4.2. In that instance, we have a changeset $S_{cs} = \{A, B, C, D, E, F\}$, and association rules $\{A\} \Rightarrow \{B, G\}$, $\{C\} \Rightarrow \{E\}$ and $\{E\} \Rightarrow \{F\}$. There are in total four predictions and $S_p = \{B, E, F, G\}$.

As three of the files in the predicted set are also in the changeset, precision is 3/4. Five files are either correctly predicted or in the left-hand side of applied rules and $S_{cov} = \{A, B, C, E, F\}$. Coverage is therefore 5/6.

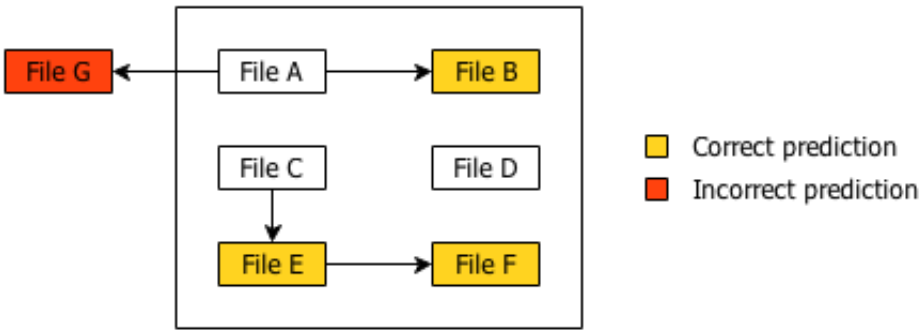

Figure 4.1. A visualization of the precision from applying change prediction on a commit. The large rectangle represents a commit and only the files that are within the bounds of the rectangle are in the commit (i.e., file "G" is *not* in the commit). Precision calculation is constituted by the colored files, where yellow and red boxes represents correct and incorrect predictions respectively. Arrows represents applied association rules. In this example, there are three correct and one incorrect prediction which results in a precision of $3/(3+1) = 3/4$.

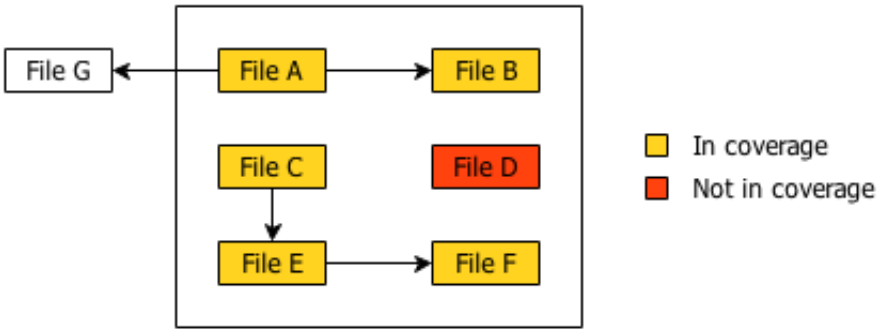

Figure 4.2. A visualization of the coverage from applying change prediction on a commit. The large rectangle represents a commit and only the files that are within the bounds of the rectangle are in the commit (i.e., file "G" is *not* in the commit). Coverage calculation is constituted by the colored files, where yellow and red boxes represents files that are in, and *not* in, the coverage respectively. Arrows represents applied association rules. In this example, out of the six files in the commit five are encompassed by the prediction, which results in a coverage of 5/6.

Finally, in order to measure how often we are able to predict, we define *feedback* as the fraction of the commits in the testing data that had one or more predictions made to them. E.g., a feedback of 0.25 means that 1/4 of the commits received one or more predictions, and, similarly, that in 3/4 of the commits we were unable do any predictions at all.

4.4 Determining Prediction Parameters

Association rule learning was controlled by specifying the minimum support, minimum confidence and the training data ratio (which we sometimes will refer to as the TDR), i.e., the fraction of the commits to be used for mining association rules. To achieve as good prediction performance as possible, which values to use for the parameters was systematically investigated for each test project.

The first parameter to be determined was minimum support. With minimum confidence and TDR set to an initial value of 0.5, the starting support value was set manually to a value as low as possible, but high enough not to make the Apriori algorithm consume too much memory (with the limit set by the available memory). The value was then incremented nine times by a small amount between 0.001-0.005, and the average precision, coverage and F-measure was calculated for each value. From the ten tests, the minimum support that generated the highest F-measure was chosen.

Secondly, minimum confidence was determined. For this the minimum support found in the previous step was used, together with the same initial TDR of 0.5. Ten tests were run with minimum confidence starting at 0.1 and incrementing by 0.1 nine times. In the same manner, the value generating the best average F-measure was chosen.

The TDR was finally determined in the same way as confidence in the previous step, except it was only tested up to 0.9 as having 100% training data would leave no room for testing.

4.5 Evaluation

The performance of the predictions was evaluated on five different open source software projects (see Table 4.1), all hosted on GitHub. In all tests, except the one described in Section 4.5.2, the commits to be analyzed were for each test repository divided into training data and testing data. Association rules were then mined from the training data and change prediction applied to each commit in the testing data. Precision and coverage were calculated for each prediction, and the average For each prediction, the precision and coverage were calculated using Eq. 4.3 and 4.4 respectively. Finally, the sought values were obtained as the arithmetic means of all values of precision and coverage, and they were used to calculate the average F-measure.

In the cases where no predictions could be done on a commit, and the cases where a changeset contained no changes, we ignored them when calculating precision and coverage. However, as another similar work handles such cases by specifying $precision = 1$ and $recall = 1$ [16], we also included this method for comparison.

In addition to the general prediction performance, two more aspects were considered: the importance of being able to retain change information for a file after it has been renamed, and whether bug fix commits are especially suited for change

Project	Main language	#commits
jQuery, web library	JavaScript	5,631
JUnit, testing framework	Java	1,880
Redis, database system	C	4,454
Ruby on Rails, web application framework	Ruby	45,290
Syncthing, file syncing software	Go	767

Table 4.1. The five projects that were analyzed along the programming languages in which they were mainly written and their number of commits at the testing date.

prediction.

Before testing could begin, minimum support, confidence and TDR were all assigned appropriate values from the process described in Section 4.4.

4.5.1 Rename Tracking

In order to evaluate the effects of Git's automatic rename tracking for change prediction, the same tests as when evaluating the general performance was run, except that the rename tracking feature was turned off when reading the test repositories. This had the implication that all file change entries of type "rename" were considered as a new file addition and did not retain potential couplings to older commits. Tests were run with the same prediction settings as the general performance tests, and results were then compared.

4.5.2 Bug Fix Commits

To investigate bug fix commits' predictive potential – especially achieving high coverage – a concrete and usable definition of a "bug fix commit" was first needed. For this purpose, it was noted that GitHub automatically closes issues when encountering any of the following keywords, combined with a number sign and an issue number, in a commit message [19]:

- close
- closes
- closed
- fix
- fixes
- fixed
- resolve
- resolves
- resolved

Because all test repositories are hosted on GitHub, using the same system to recognize which commits close bugs seemed like a natural idea. The definition of a bug fix commit was therefore settled on: *a commit which message contains any of GitHub's issue-resolving keywords plus a number sign, separated by a space*

character. E.g., a commit message containing "resolves #123" would count as a bug fix commit whereas just "resolves" would not.

Commit Filtering

Two different approaches were taken to investigate bug fix commits' predictive potential. The first approach, which was named *commit filtering*, was to apply a filter to the commits at the repository reading stage and only extract commits matching the definition. Thereafter, the prediction was evaluated in the same manner as the general performance tests, enabling us to compare the performance of only having bug fix commits versus using the whole repository.

As this filtering had the likely consequence of significantly reducing the sizes of the training and testing sets, data mining parameters had to be reconfigured with the process in Section 4.4.

Coverage Range Matching

The second approach was to specifically investigate if single commit predictions would have a higher coverage for bug fix commits than other commits. In this case, the most recent 1,000 commits were chosen as testing commits for each test project, with the exception of Syncthing in which the newest 600 commits was used, as Syncthing in total had fewer than 1,000 commits. Then, for each test commit, all commits older than it were used to mine association rules (see Figure 4.3). This means that the association rule mining phase was initiated before each commit prediction. The individual coverage for each test commit was then saved and imported as a table into an SQLite database, together with the whole repository, for further analysis.

Prediction parameters used in this particular test were the same as in the general performance test, except that minimum support for jQuery and JUnit was increased to 0.005 and 0.01 respectively to not run out of memory.

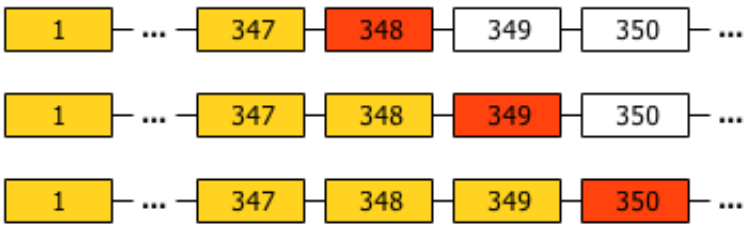

Figure 4.3. A visualization of the continuous association rule mining and prediction that was used when determining the fraction of bug fix commits receiving high coverage predictions. Boxes represent numbered commits, where higher numbers corresponds to newer commits. Each of the three rows represents a prediction iteration where the yellow boxes are commits used for association rule mining, and red boxes commits on which change prediction is applied.

We defined a “high coverage commit” as a commit having a coverage higher than or equal to 0.8. Commits with empty changesets are represented by `NaN` and these were removed from all tests. By altering the SQL statement slightly the total number of bug fix commits and the total number of high coverage commits respectively could be obtained.

Finally, the fraction of bug fix commits having high coverage was compared to the fraction of non-bug fix commits, as well as all commits, having high coverage.

Chapter 5

Results

This chapter lists the results of the experiments described in Section 4.5.

Section 5.1 discusses the results from the tests that focus on, as the section name suggests, evaluating the general prediction performances for the five test projects. Section 5.2 examines the potential benefits of the VCS's automatic rename tracking functionality. Lastly, Section 5.3 discusses the results from the tests investigating the predictive possibilities of bug fix commits.

Test parameters minimum support, minimum confidence and TDR have been individually configured for each test project as in the procedure described in Section 4.5, and the same parameters are used in each test setup if not specified otherwise. Test settings can be seen in Table 5.1.

All tests were run on a laptop computer with a 1.9 GHz Intel i7-3517U CPU and 6 GB internal memory, running Windows 8.1.

5.1 General Prediction Performance

Table 5.1 shows the average precision, coverage and F-measure together with the feedback from performing change prediction on a portion of each test project's commits. Numbers in parentheses are the results from defining $precision = 1$ for a commit when no prediction could be made and $coverage = 1$ when a changeset is empty.

jQuery and Syncthing sticks out with an F-measure of 0.47 and 0.41 respectively while the other three test projects span from 0.18 to 0.29. The average feedback is 0.41, which means that predictions, on average, were made to 2/5 of the test commits. The difference between the two different kinds of precision measurement is quite high which goes hand in hand with the low feedback.

5.2 Rename Tracking

Table 5.2 shows the prediction results from evaluating rename tracking.

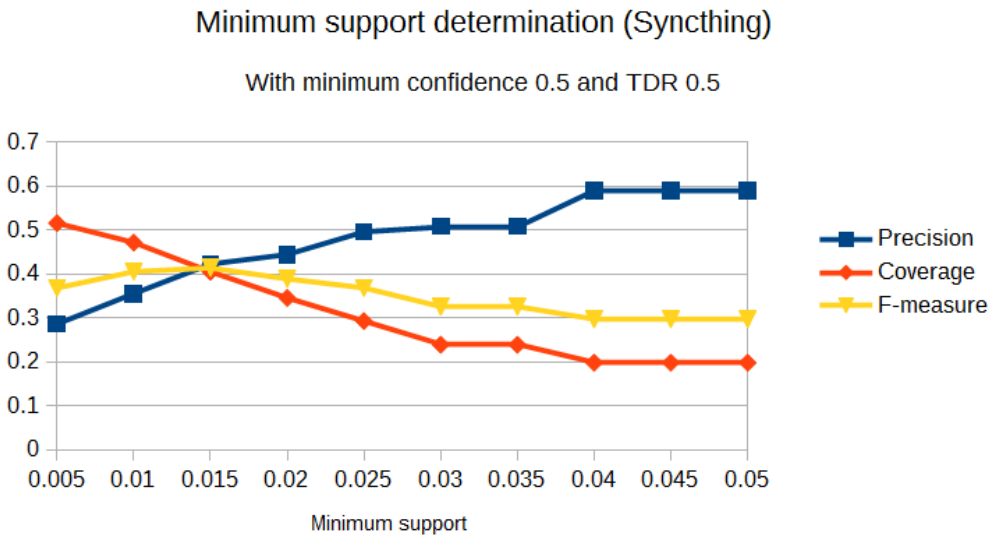

Figure 5.1. Shows how precision, coverage and F-measure varies with minimum support, from applying change prediction to the Syncthing repository. Precision seems to increase and coverage decrease until minimum support reaches 0.04. Highest value for F-measure is achieved for minimum support 0.015.

Results show that F-measure is slightly lower for each of the test projects in comparison to the general performance test, indicating that retaining file history for renamed files has a positive impact on change prediction performance. Both tests achieve an average feedback of 0.41, although it differs slightly between projects. Ruby on Rails receives an especially large performance drop in this test as F-measure and feedback both drop by roughly 20%.

5.3 Bug Fix Commits and Coverage

This section contains the results from investigating the hypothesis that bug fix commits yield higher prediction performance than other commits in general.

Section 5.3.1 shows the results from filtering the commits on their commit messages matching certain keywords, and mining association rules and applying change prediction to them only. Section 5.3.2 shows the results from mining association rules, calculating the coverage for each commit to be analyzed, and then investigating how large a fraction of the bug fix commits also have a high coverage.

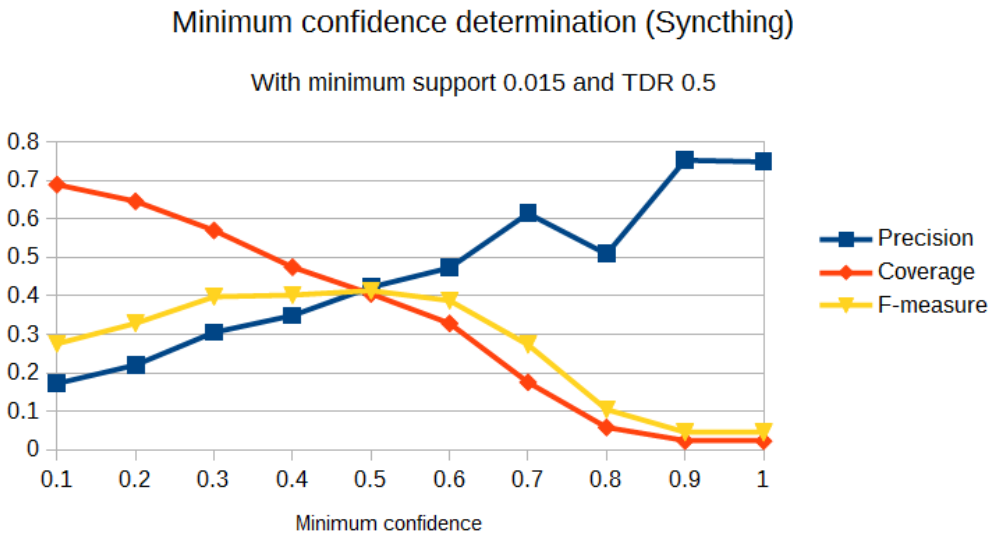

Figure 5.2. Shows how precision, coverage and F-measure varies with minimum confidence, from applying change prediction to the Syncthing repository. Precision seems to steadily increase, and coverage steadily decrease, as minimum confidence increases. Precision unexpectedly drops for confidence 0.8, to increase again. Highest value for F-measure is achieved for minimum confidence 0.5.

5.3.1 Commit Filtering on Keywords

Table 5.3 shows the results from mining association rules and performing change prediction only on bug fix commits. The test is done in the same way as the test in Section 5.1, except for being applied on a subset of commits in each test project. Redis was excluded from the test due to not being able to do a single prediction, an issue due to only having ten commits matching the bug fix definition.

F-measure is higher in each of the four tests when compared to using all commits, and the average precision, coverage, F-measure and feedback are also all higher. For jQuery and Ruby on Rails the differences are especially notable; for jQuery, F-measure is roughly 50% higher while for Ruby on Rails it has more than doubled.

5.3.2 Keyword Matching and High Coverage

Table 5.4 shows a comparison of the fractions of bug fix commits that have high coverage predictions to non-bug fix commits and all commits.

Bug fix commits seem to be more prone to receive high-coverage predictions than non-bug fix commits, looking at the average and the first three projects. For the last two projects however, the opposite result can be observed, albeit only slightly. Comparing non-bug fix commits to all commits reveals no obvious differences.

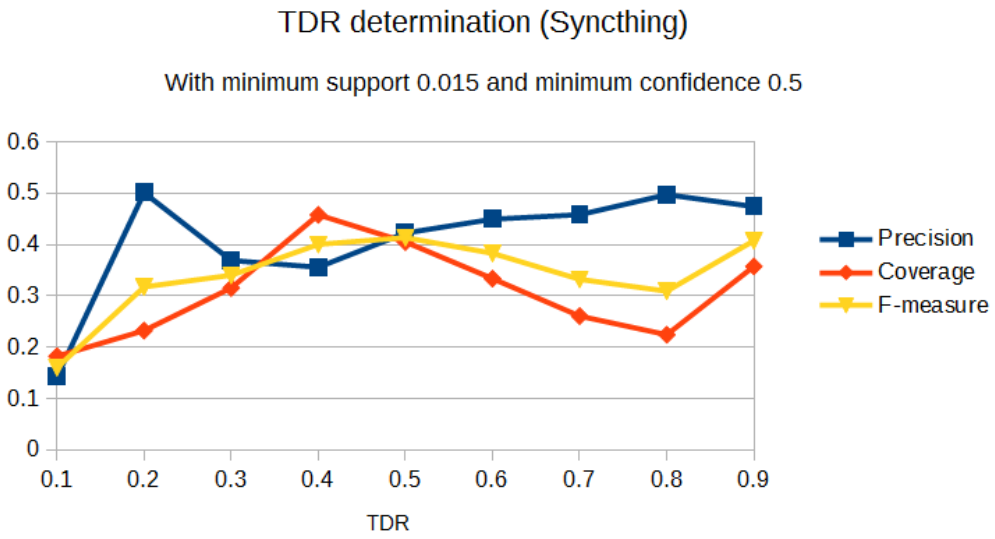

Figure 5.3. Shows how precision, coverage and F-measure varies with TDR, from applying change prediction to the Syncthing repository. Precision seems to first rapidly increase to later stabilize, and has its peak at TDR 0.2. Coverage slowly increases and peaks at TDR 0.4. Highest value for F-measure is achieved for TDR 0.5.

Project	P_{avg}	C_{avg}	$F1$	Fb	$supp$	$conf$	tdr
jQuery	0.48 (0.78)	0.46 (0.54)	0.47 (0.64)	0.43	0.0005	0.5	0.5
JUnit	0.18 (0.65)	0.35 (0.39)	0.23 (0.49)	0.43	0.005	0.5	0.5
Redis	0.25 (0.69)	0.35 (0.37)	0.29 (0.48)	0.41	0.015	0.4	0.3
Ruby on Rails	0.15 (0.74)	0.25 (0.34)	0.18 (0.47)	0.31	0.002	0.5	0.4
Syncthing	0.42 (0.71)	0.41 (0.42)	0.41 (0.53)	0.5	0.015	0.5	0.5
Average	0.29 (0.71)	0.36 (0.41)	0.32 (0.52)	0.41			

Table 5.1. Shows the average precision, coverage, F-measure and feedback, along with the prediction settings used for each project. Numbers in parentheses are the results from defining $precision = 1$ for commits where no predictions could be done and $coverage = 1$ for commits containing no changes, while numbers outside parentheses are from excluding said commits. jQuery and Syncthing achieve the highest F-measure while Ruby on Rails achieves the lowest. Feedback is hovering between 0.4 and 0.5 for all projects except Ruby on Rails. The difference between the two types of precision calculations is significant while the difference between coverage calculations is marginal.

Project	P_{avg}	C_{avg}	$F1$	Fb
jQuery	0.5	0.44	0.47	0.41
JUnit	0.14	0.39	0.2	0.5
Redis	0.24	0.34	0.28	0.4
Ruby on Rails	0.12	0.18	0.14	0.24
Syncthing	0.34	0.38	0.36	0.5
Average	0.27	0.35	0.29	0.41

Table 5.2. Shows the average precision, coverage, F-measure and feedback from evaluating change prediction on Git repositories without preserving change histories for renamed or moved files. Prediction settings used were the same as for the "general performance" evaluation. Compared to the results in Table 5.1, all measure averages from this test are slightly lower, except feedback which stays unchanged.

Project	P_{avg}	C_{avg}	$F1$	Fb	$supp$	$conf$	tdr
jQuery	0.63	0.72	0.67	0.72	0.002	0.5	0.2
JUnit	0.3	0.26	0.28	0.31	0.069	0.1	0.5
Redis	–	–	–	–	–	–	–
Ruby on Rails	0.4	0.36	0.38	0.45	0.0015	0.5	0.3
Syncthing	0.42	0.46	0.44	0.61	0.027	0.5	0.5
Average	0.44	0.45	0.44	0.52			

Table 5.3. Shows the average precision, coverage, F-measure and feedback from evaluating change prediction on *bug fix commits*. Redis had to be excluded due to not being able to produce a single prediction. Compared to the evaluation of all commits in Table 5.1, all measures are higher for the bug fix commits with the exception of coverage and feedback for JUnit. The average F-measure is almost 40% higher, and in the case of Ruby on Rails it has increased by more than 100%.

Project	Bug fix commits	Non-bug fix commits	All
jQuery	0.46	0.29	0.33
JUnit	0.4	0.23	0.24
Redis	0.38	0.18	0.18
Ruby on Rails	0	0.03	0.03
Syncthing	0.29	0.3	0.3
Average	0.3	0.2	0.21

Table 5.4. A comparison of the fractions of bug fix commits, non-bug fix commits and all commits, that received *high-coverage predictions*, i.e., predictions that covered 80 to 100% of the files. For most projects, the amount of high-coverage predictions is, percentage wise, significantly higher in bug fix commits. The exceptions are Ruby on Rails and Syncthing where they are roughly the same.

Chapter 6

Discussion

In this chapter the results are discussed and compared with existing and similar works.

6.1 General Performance

The confidence that resulted in the highest value for F-measure is similar for all projects: about 0.5. As a higher confidence generally results in higher precision and lower recall (in this case, coverage), this result comes as no surprise. The F-measure is favored by simultaneously having both high precision and coverage, which is probably what would be expected from a confidence in the vicinity of 0.5.

That the best TDR also lies close to 0.5 for every project is a bit harder to explain however. The likelihood that the testing data contains files not present in the training data increases with the size of the testing data, and such files would naturally be impossible to predict. One explanation could be that the minimum support was specified *before* the TDR; initially setting the TDR to a higher value than 0.5 could possibly have produced even better results.

The two projects jQuery and Syncthing receive noticeably better scores than the other projects. It would be interesting to find out if they share any similar traits that make them especially beneficial to perform change prediction on. Sadly, an extensive analysis of this is out of scope for this project and we may only speculate. They do not have a similar number of commits, they are written in two different and quite unrelated programming languages and have different purposes. The similarities may instead lie in how they do commits. For example: smaller, incremental commits that focus on fixing one particular issue ought to be more useful for predictive purposes. Another reason could be that they simply have a higher amount of coupling, which also comprises structural coupling, as file-level dependencies found by mining association rules are observed tendencies and may not inherently consist of logical coupling only.

In [16] T. Zimmermann et al. also investigated file-level logical coupling and our results ought to be compared with theirs. Our test strategies are not completely

identical with theirs however, which should be kept in mind. The best approach is probably to compare the results of the coarse granularity *Navigation* scenario; using a support count of one and a confidence of 0.1 for all of their eight test projects, they achieved an average precision and recall of 0.29 and 0.44 respectively, with a feedback ratio of 0.82. As they counted a predictionless commit as $precision = 1$, and an empty commit as $recall = 1$, their results should be compared to the ones in parentheses in Table 5.1.

We can observe that the average precision in our tests is a bit more than double, while the feedback is about half that of their results. This conforms to the notion that one can either have precise predictions or many predictions, but not both, as stated in [16]. Despite coverage being more generous than recall, they achieve a higher recall than we achieve coverage, something that is likely to be related to their high feedback and low precision.

Determining which test result is the more satisfactory is not entirely trivial, as they consist of several different types of measures. We can eliminate precision and recall by calculating the F-measure: a precision and recall of 0.29 and 0.44 gives an F-measure of 0.35. Our test has a 50% higher F-measure, but only half the feedback ratio.

In [15], another study on file-level coupling, A. Ying et al. achieved an average precision and recall of about 0.35 and 0.15 respectively when analyzing the Eclipse project, and 0.5 and 0.25 respectively when analyzing the Mozilla project. Calculating the F-measure gives us 0.21 and 0.33, which is, similarly to [16], slightly lower than our results. Nonetheless, there are a few prominent differences in our test setups. They have chosen to favor precision over recall and have not included the feedback ratio, making comparison a somewhat intricate endeavor.

All in all, the conclusion we can make is that our results and the results from [16] and [15] are roughly similar.

6.2 Rename Tracking

Discarding file change histories for renamed files seems to lead to a slightly lower prediction score in comparison to retaining them, as can be observed in Table 5.2. This result indicates that coupling typically persists across rename events and that using a VCS that supports tracking of renamed files, such as Git, is marginally beneficial if one is to do change prediction.

In the cases of jQuery and JUnit, either precision or coverage is higher with rename tracking turned off. This could seem counter-intuitive at first as one may be inclined to believe that more preserved coupling should generate a greater number of rules, which could be beneficial for the prediction. There are some cases where this is not necessarily true though, for example if the rename event is part of a refactoring that weakens or removes some or all of the coupling related to the renamed file. Another example is that the coupling has been "faulty" from the start and led to bad predictions. In both cases, disregarding the file history when the file is renamed

could lead to a better prediction score.

6.3 Bug Fix Commits

The higher prediction scores for bug fix commits can be explained by the fact that they are likely to, to a higher degree, involve files that have previously been altered together, and are unlikely to introduce any new files. This is in contrast to commits that introduce new features, which would be more difficult to predict.

This result is similar to the findings made in [16], which indicate that commits that only contain alterations (and no additions or deletions), in their paper referred to as *maintenance commits*, have higher recall and feedback. One difference seems to be apparent however: in their case, only recall and feedback saw benefit from focusing on maintenance commits, while precision stayed roughly the same. Our results, on the other hand, suggest that *all* measures increase. More so, average precision increased by about 50%.

While the higher coverage can be explained by the lack of file additions, giving a clear reason to why precision also increased is more difficult. One hypothesis is that bug fix commits perhaps generally have a lower cardinality and are more focused on solving one specific task than other commits. This could imply fewer applicable association rules and, possibly, also a higher precision.

Matching high coverage and bug fix commits, as seen in Table 5.4, further suggests that bug fix commits are more likely to receive high coverage predictions than other commits.

It shall be noted that the definition of a maintenance commit and bug fix commit are not entirely identical. Zimmermann et al. define a maintenance commit as a commit that only have fine-grained alterations, i.e., a commit that does not have any added or deleted methods or fields. This level of detail is of course unobtainable at the file-level granularity that this project utilizes.

6.4 Threats to Validity

Five different software projects have been analyzed, and while they have been chosen to not be too similar – to differ in programming languages, application areas and size – they cannot be guaranteed to be representative for *all* software projects. Two common traits among the projects are that they are all relatively young projects and that they are hosted on GitHub.

One problem with this technique is the difficulty of choosing appropriate prediction parameters as some of them are codependent. E.g., support is dependent on the cardinality of the training data, which in turn is determined by the TDR. As all parameters need to first have been assigned a value in order to do a prediction evaluation; they cannot each be tested in isolation. Therefore, when the "best" value for minimum support has been determined, it is unlikely that changing the TDR will have any significant positive effect.

Moreover we can not guarantee that the software used to perform the tests is completely correct and bug free, especially not the part that we wrote ourselves. In order to minimize the risk of our program becoming a victim of unpleasant defects we kept to well-known and well-tested technologies, and we also offloaded the most error critical part to arules [18]. Still, this is no guarantee.

Finally, our definition of a "bug fix commit" does not necessarily conform exclusively to a bug fixing commit, as an "issue" in an issue tracking system in some cases could refer to something else than a bug, e.g., an enhancement suggestion. Besides, we cannot be absolutely sure that the GitHub issue tracking system was actively used for each project, even though they all contained numerous issues.

Chapter 7

Conclusions

In this thesis, we have extracted file-level logical coupling from the Git version histories of five different open source software projects, and evaluated its usefulness for predicting changes. This chapter highlights the most interesting results, presents a conclusion and makes suggestions for future work.

7.1 Main Results

Results showed an average value of the F-measure of 0.32 and an average feedback of 0.41, which roughly coincides with previous studies [15][16].

Results also revealed that bug fix commits are slightly more likely to receive accurate predictions, receiving an average F-measure of 0.44 and an average feedback of 0.52. This conforms to the results of evaluating so called *maintenance* commits in a previous study [16], although it differs in that precision increased in our case while in their case it remained unchanged. This difference could stem from the distinct definitions; "bug fix" commits are commits that fix one or more issues in the issue tracker, while "maintenance" commits are commits that does not contain any additions or deletions of entities.

Investigation of the preservation of change history when renaming or moving a file showed it to be of mild significance for change prediction, as disregarding file history in such cases only resulted in slightly lower prediction scores; across the five projects, the average F-measure decreased from 0.32 to 0.29 while the feedback stayed unchanged.

7.2 Conclusion

While the prediction results on average were slightly higher in the cases where renamed or moved files were taken into consideration in comparison to when they were not, the improvement can not be considered significant enough to draw the conclusion that modern VCSs are better suited for change prediction. For a couple

of the test projects, the feedback or F-measure was in fact slightly lower or equal with rename tracking turned on.

Similarly, in the case of bug fix commits, the prediction results were on average slightly higher in the cases where only bug fix commits were taken into consideration in comparison to when they were not. Here, however, the difference is more distinct and suggests that mining rules from, and doing prediction on, only the subset of bug fix commits of a project is better than using all commits if one is out to achieve good prediction.

What may be the reasons for the unsatisfying results? Could we have performed the study in a way such that the results would have been more conclusive?

Regarding bug fix commits, one potential source of error is that the issue referenced by a bug fix commit may be a feature addition, as users can file "enhancement requests". Our assumption was that the issue tracking system for a mature project would contain mostly bug related issues as more and more features are implemented, but it is possible that the outcome would have been different had we filtered only the commits that referenced bug fix related issues.

Regarding rename tracking, it seems that the event of a file being moved or renamed was not occurring frequently enough in the analyzed projects to make rename tracking a vital component in change our prediction. It is possible that it could be of greater importance in a project that is refactored heavily and often, though we suspect that renaming or moving a file is a much rarer type of refactoring than, for example, breaking out a new function or class.

7.3 Future Work

It would be interesting to investigate if filtering on commits that fixes issues that actually are bugs could give even better results. To do this, one have to be able to link commits to their respective issues and parse the issue in question. Besides, a bug fixing commit is not necessarily absent of any new file additions, as a developer not following Git's best practices could possibly make a commit *both* fixing a bug and adding a new feature. For these cases, combining the properties of bug fix and maintenance commits to find out the potential of bug fixing commits not adding new files could perhaps result in even better prediction scores.

Bibliography

[1] C. Brindescu, M. Codoban, S. Shmarkatiuk and D. Dig, "How Do Centralized and Distributed Version Control Systems Impact Software Changes?", *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, pages 322-333, 2014.

[2] S. Chacon and B. Straub, "Pro Git, Second Edition", *Apress*, pages 27-33, 2014.

[3] V. Chandola, A. Banerjee and V. Kumar, "Anomaly Detection: A Survey", *ACM Computing Surveys (CSUR), vol. 41, no. 3*, page 13, July 2009.

[4] D. Čubranić, G. Murphy, J. Singer and K. Booth, "Hipikat: A Project Memory for Software Development", *IEEE Transactions on Software Engineering, vol. 31, no. 6*, pages 446-465, June 2005.

[5] U. Fayyad, G. Piatetsky-Shapiro and P. Smyth, "From Data Mining to Knowledge Discovery in Databases", *AI Magazine, vol. 17, no. 3*, pages 37-54, 1996.

[6] H. Gall, K. Hajek and M. Jazayeri, "Detection of Logical Coupling Based on Product Release History", *International Conference on Software Maintenance*, pages 190-198, Nov. 1998.

[7] H. Gall, M. Jazayeri and J. Krajewski, "CVS Release History Data for Detecting Logical Couplings", *International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 13-23, Sept. 2003.

[8] D. Hand, H. Mannila and P. Smyth, "Principles of Data Mining", *MIT Press*, pages 1-4 and 157-160, 2001.

[9] P. Simon, "Too Big to Ignore: The Business Case for Big Data", *Wiley*, page 89, March 2013.

[10] G. Lionetti, "What is Version Control: Centralized vs. DVCS", *Atlassian Blogs*, Feb. 2012, `http://blogs.atlassian.com/2012/02/version-control-centralized-dvcs/`, accessed 2015-01-28.

[11] S. Negara, M. Vakilian, N. Chen, R. Johnson and D. Dig, "Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?", *ECOOP 2012 – Object Oriented Programming*, pages 79-103, 2012.

[12] R. Robbes and M. Lanza, "Versioning Systems for Evolution Research", *International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 155-164, Sept. 2005.

[13] S. J. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach, Third Edition", *Prentice Hall*, pages 694-695, 2010.

[14] Z. Xing and E. Stroulia, "Data-mining in Support of Detecting Class Co-evolution", *16th International Conference on Software Engineering and Knowledge Engineering (SEKE '04)*, June 2004.

[15] A. Ying, G. Murphy, R. Ng and M. Chu-Carroll, "Predicting Source Code Changes by Mining Change History", *IEEE Transactions on Software Engineering, vol. 30, no. 9*, pages 574-586, Sept. 2004.

[16] T. Zimmermann, P. Weißgerber, S. Diehl and A. Zeller, "Mining Version Histories to Guide Software Changes", *IEEE Transactions on Software Engineering, vol. 31, no. 6*, pages 429-445, June 2005.

[17] About, `http://sourceforge.net/about`, accessed 2015-03-08.

[18] arules: Mining Association Rules and Frequent Itemsets, `http://cran.r-project.org/web/packages/arules/`, accessed 2014-09-01.

[19] Closing issues via commit messages · GitHub Help, `https://help.github.com/articles/closing-issues-via-commit-messages`, accessed 2014-09-27.

[20] The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things, `http://idcdocserv.com/1678`, accessed 2015-02-16.

[21] JGit, `http://www.eclipse.org/jgit/`, accessed 2014-09-01.

[22] Press – GitHub, `https://github.com/about/press`, accessed 2015-04-12.

[23] The R Project for Statistical Computing, `http://www.r-project.org/`, accessed 2014-09-01.

[24] Rserve – Binary R server, `http://rforge.net/Rserve/`, accessed 2014-09-01.