Big Commit Analysis

Towards an Infrastructure for Commit Analysis

Master Thesis

Andreas Hohler
from
Lohn-Ammannsegg SO, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

15. Januar 2018

Prof. Dr. Oscar Nierstrasz
Haidar Osman

Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Abstract

Developers commit changes to the code base of a certain project in order to, for instance, fix bugs, add features, or refactor the code. In empirical studies, researchers often need to link commits with issues in issue trackers to audit the purpose of code changes. Unfortunately, there exists no general-purpose tool that can fulfill this need for different studies. For instance, while in theory each commit should serve one purpose, in practice developers may include several goals in one commit. Also, issues in issue trackers are often miscategorized.

We present BICO (BIg COmmit analyzer), a tool that links the source code management system with the issue tracker. BICO presents information in a navigable form in order to make it easier to analyze and reason about the evolution of a certain project. It takes advantage of the fact that developers include issue IDs in commit messages to link them together. BICO also provides dedicated analytics to detect *big commits*, *i.e.,* multi-purpose and miscategorized commits, using statistical outlier detection. In an initial evaluation, we use BICO to analyze bug-fix commits in Apache Kafka, where our tool reports 9.6% of the bug-fixing commits as miscategorized or multi-purpose commits with a precision of 85%. This high precision demonstrates the applicability of the outlier detection method implemented in BICO. A further case study with Apache Storm shows that the precision of detecting multi-purpose commits can vary between projects. In addition, BICO also comes with a built-in metric suite extractor for calculating change metrics, source code metrics and defect counts.

Contents

1 Introduction

Developers commit source code changes for many reasons, *e.g.,* to fix bugs, add features, or clean up the code. The purposes of these commits are usually documented in the commit messages themselves. Developers also often include issue IDs in commit messages to link their commits with the issues they resolve. This ad-hoc style of linking source code management systems with issue trackers motivates researchers to mine both repositories and deduce knowledge from the evolution of systems, like discovering bug-fix patterns [1][2][3], predicting changes in the code [4][5], and building datasets for bug prediction [6]. Such studies face three main challenges:

Linking commits with issues Although many techniques have been proposed in the past to approach this problem (*e.g.,* [7]), there exists no general-purpose tool that can be used off the shelf. Each study has its own implementation.

Finding clean commits Ideally, commits should not be big and each commit should serve one purpose. However this is not necessarily true in practice. There are commits that serve multiple purposes at the same time and vary in size as in Figure 1.1. Studies can produce more reliable results if they rely on clean commits, *i.e.,* uni-purpose commits.

For example, a refactoring process can touch many files and change many lines of code, but a *NullPointerException*[1] fix touches only one file.

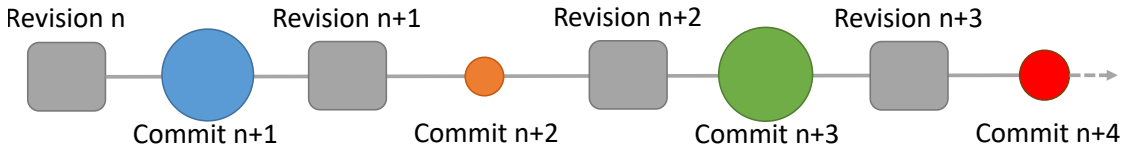


Figure 1.1: Evolution of a software project with different commit sizes and purposes

[1] `https://issues.apache.org/jira/browse/FLUME-2672`

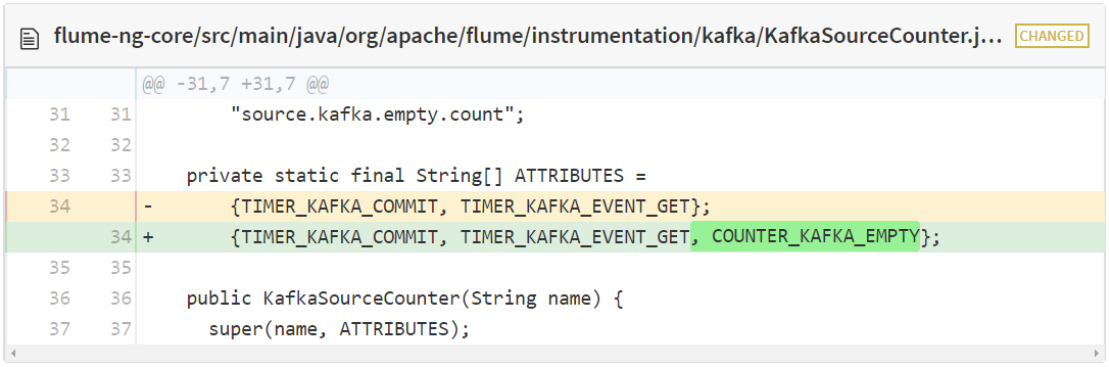


Figure 1.2: A NullPointerException fix in Apache Flume

If a commit has multiple purposes, it is a so called *big commit*. Herzig *et al.* [29] describe a big commit as a *tangled commit*. We stick to the term *big commit* in this work. In the Apache Lucene project we found a commit that was classified as bug by the issue LUCENE-6271[2]. The assigned issue states that a function does not return consistent values. This sounds like some small bug. But in the commit 3'945 lines of code and 73 files were changed. Clearly, this is not just a bug fix but rather a big commit. Examining the commit itself reveals that the commit has many purposes including several new features, optimizations and bug fixes. The commit message on the master branch says "sync up with trunk". We now know that is not particularly a big commit, but also a *merge* commit. We should differentiate between those types of big commits in the analysis.

Categorizing issues correctly It has been shown previously that issues in issue trackers are sometimes miscategorized [8][9]. This threatens the external validity of the studies that rely on accurate categorization.

To address these challenges, we present BICO, a general-purpose tool that links commits and issues, and provides further analytics to detect suspicious commits that either combine multiple purposes or are miscategorized. For instance, issue number 6271 in Apache Lucene is actually a refactoring commit. Eliminating such commits from analysis, or at least the awareness of their controversy, improves the reliability of further empirical studies in software evolution. BICO implements statistical outlier detection to detect big commits like this one. An initial evaluation to extract and analyze bug-fix commits in Elasticsearch shows that BICO could categorize 1'489 commits as fixes, 7% of which are detected as big commits with a precision of 85%.

BICO represents a first step to build a general-purpose infrastructure for analyzing software evolution. To demonstrate its usefulness, we built a metric suite extractor on top of BICO for calculating change metrics [10], source code metrics [11], and defect counts [7] for any analyzed system at the point in time of any commit. BICO, with its big commit analysis, provides a usable infrastructure that facilitates this metric extraction.

[2] https://issues.apache.org/jira/browse/LUCENE-6271

2
Related Work

Mockus and Votta [13] categorize the changes of software based on the textual content in transaction log messages in the Extended Change Management System (ECMS) [14].

Śliverski *et al.* [7] identify the commits that induce fixes in the future. They start from the bug report in an issue tracker, then navigate to the commit that fixes the issue, identify the changed code in that commit, and finally track that changed code to the commit that introduces it. Analyzing Eclipse and Mozilla, Śliverski *et al.* reveal that the average number of changed files in a fix commit tends to be small: 2.73 in Eclipse and 4.39 in Mozilla. Purushothaman and Perry [15] analyze small changes in software using change and defect history data. They find that nearly 10% of changes are one-liners and the maximum number of changes are adaptive (*i.e.,* new features). After a manual inspection of 374 bugs from three systems, Lucia *et al.* [12] find that bug fixes that span more than five files are very rare (7% in Rhino, 1% in AspectJ, 10% in Lucene). Herzig *et al.* [9] manually examined more than 7'000 issues from issue trackers of five open-source projects. They report that between 37% and 47% of issue reports are wrongly typed in issue trackers. This type of study can benefit from an off-the-shelf tool such as BICO.

Fischer *et al.* combine data from Mozilla CVS and the Bugzilla issue tracker into one database called the release history database as a part of a software evolution analysis framework [16]. Then they use this combination of data sources to pinpoint and track features in the source code and reveal relationship between features in Mozilla [17]. BICO provides similar functionality but on a wider scale where many issue trackers are supported and any git repository can be analyzed.

Dallmeier and Zimmermann [18] propose `iBUGS`, a tool that extracts bug localization data semiautomatically from software change history. Using patterns like "Fixed 1234" or "Bug #1234" in commit messages, they discover bug-fix commits, but without linking them to issue trackers. The authors made many datasets available using `iBUGS`. BICO provides a step further by linking the commit messages to issues in issue trackers.

Begel *et al.* develop a framework called *Codebook* [19][20][21] that extracts data from several types of software repositories and combines them into people-artifacts graphs. However, *Codebook* is tailored to the infrastructure at Microsoft and cannot be used in other contexts. BICO on the other hand is publicly available and can work with most of open-source project setups.

Rosen *et al.* present Commit Guru [22], a tool that can provide commit analytics for any publicly accessible git repository. The main goal is to identify risky commits (*i.e.,* bug-introducing commit)

[23][24]. Commit Guru calculates commit-level metrics (*e.g.,* number of modified files, Age from last change) and uses them to predict how risky every commit is. However, using the approach suggested by Hindel *et al.* [25], Commit Guru relies only on the commit messages in order to categorize commits. It also assumes that each commit belongs to one category. BICO is different from Commit Guru in that it links issues and commits, identifies big commits, and extracts change and source code metrics at the Java file level.

Hindle *et al.* [25] define the term *Large Commits* as the commits that include large numbers of files. Hindle *et al.* carried out a case study on 9 open source projects, and manually analyzed the 1% of the commits that contain the largest number of files, any files and not only source code files. They concluded that large commits are likely to be perfective while small commits are likely to be corrective. In this paper we define big commits differently, as the commits that are miscategorized or serve multiple purposes. We also categorize large commits, rather than just listing all of them.

Bachmann *et al.* introduced Linkster [26], a tool that connects version control history and bug report history to identify defect-fix commits. The main intent of Linkster is to help developers and researchers navigate and annotate commits. BICO uses similar analysis for linking commits and issues. BICO also can be used for exploration but is mainly intended for analysis. BICO identifies categories of commits, identifies big commits, and extract software metrics the analyzed system at any commit.

Although the techniques for linking commits with issues are already explored in the literature, there exists no tool or implementation that can be used. BICO aims at filling this gap and facilitating reproducible empirical research in software engineering.

3 Technical implementation

3.1 Architecture

Under the hood, BICO is mainly a batch job tool. Starting from the UI, when a user adds a new project to be analyzed, BICO starts a batch job in the backend that does the actual heavy lifting. A Spring batch job can have several steps and each step has three components: the reader, the processor and the writer. Each batch job in BICO consists of two steps. In the first step BICO clones the repository, parses the commits, extracts issue IDs from commit messages, and saves the results with the help of the Hibernate ORM and Java Persistence API in the PostgreSQL[1] database. As soon as the first step is successfully completed, the second step starts, where BICO retrieves all the issues for all the IDs extracted in the previous step and links them with the commits in the database. In the third and final step, analytics are generated and big commits are detected. After this step, the user can explore the extracted data via the web interface. BICO also provides a means to control the batch jobs themselves. Users can stop, restart, pause, and resume the jobs at anytime. The batch job backend is implemented in Spring Batch and the web front-end is implemented in Spring MVC[2].

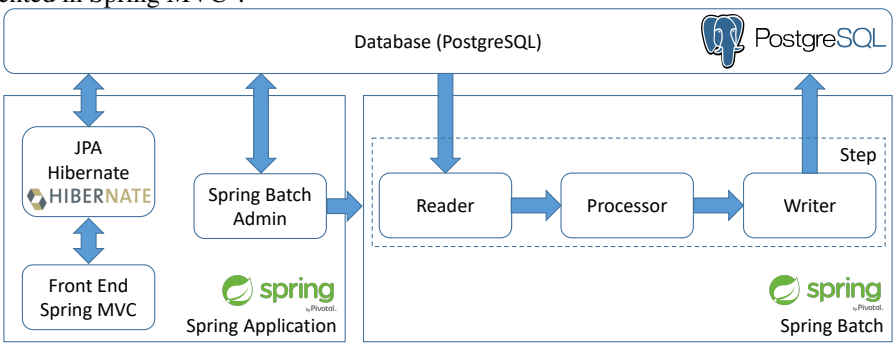


Figure 3.1: The BICO tool architecture

[1] https://www.postgresql.org
[2] https://spring.io

3.2 Database

All data from BICO is stored in a PostgreSQL[3] database. This is a very valuable part of the tool since the database can be queried separately without interacting with the UI.

The main schema is shown in Figure 3.2. The project table is the entry point with a 1:N connection to commits. We would like to point out that commits and issues have an N:N relationship. This reflects the reality in software development, where in one commit several issues could be addressed, or several commits could fix a single issue. Another benefit from this structure is that we will not have any duplicated issue entries and therefore the batch job does not have to process them multiple times. The commit table has many 1:N relationships, *e.g.,* to every metrics table and to commit files.

There exist a few other tables starting with "batch_" used for Spring Batch and are not part of the main data model.

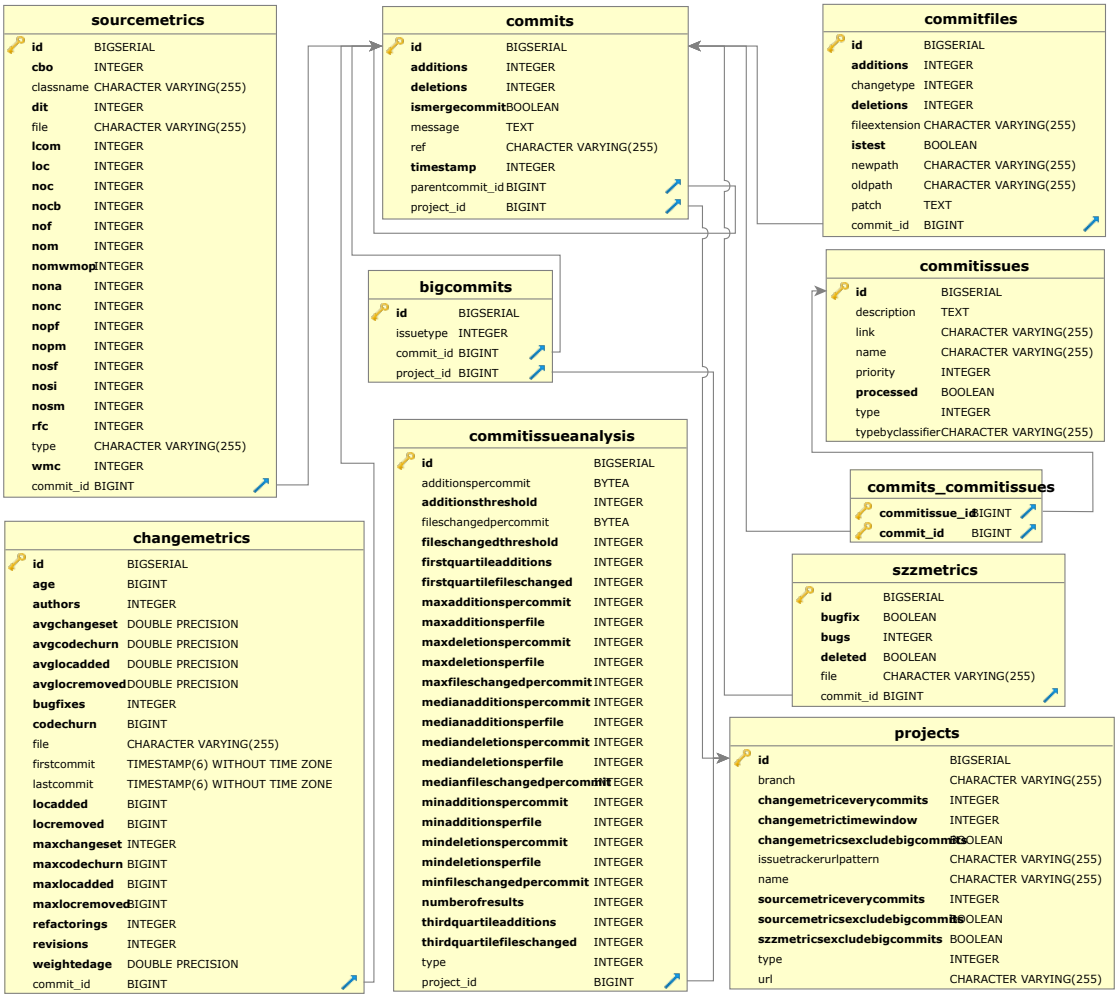


Figure 3.2: The main BICO database schema

[3] https://www.postgresql.org

3.3 Batch processing

The Spring Batch module is essential for managing tasks and jobs. A job consists of at least one step, where every step consists of a reader, processor and writer, or just of a tasklet.

An Item Reader is the entity of a step that reads data into the spring batch application from a particular source. It reads one item a time and is represented by the interface ItemReader<OutputObject>.

An Item Processor is a class which contains the processing code, which is executed on every read record and processes the data. When the given item is not valid it returns null (and the step terminates), else it processes the given item and returns the processed result. The interface ItemProcessor<InputObject,OutputObject> represents the processor.

An Item Writer is the part of a step that writes data from Spring Batch application to a particular destination and writes a specific amount of items (chunks) a time and is represented by the interface IteamWriter<InputObject>

A tasklet acts as a processor, when no reader and writer are given and it processes only a single task.

3.3.1 Repository mining

This procedure is organized as a job with three steps.

Step 1 clones the repository, reads all commits and parses the issue identifier of the used issue tracker. For each commit, BICO keeps track of the meta data such as the git reference, number of added lines, number of deleted lines, and the patch of each changed file in that commit. Figure 3.3 visualizes this step.

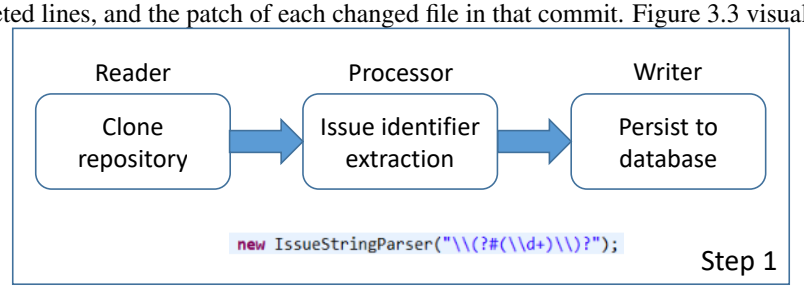


Figure 3.3: Repository mining: cloning the repository

Step 2 uses the issue identifiers to link the commits with issues from the tracker as seen in Figure 3.4. Currently, there are two basic implementations of getting information from the trackers: retrieval with HTTP requests and with the GitHub API.

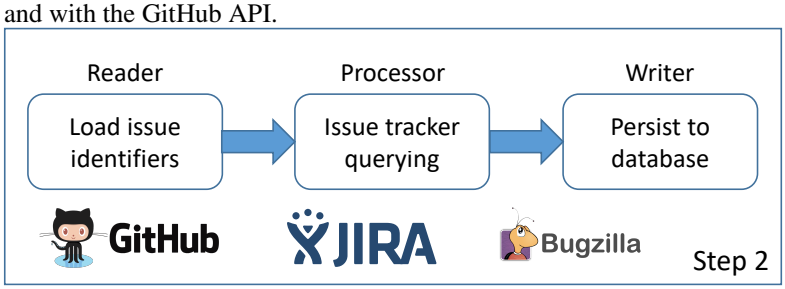


Figure 3.4: Repository mining: linking issues

Step 3 counts some statistic variables and detects big commits Figure 3.5.

Figure 3.5: Repository mining: Analysis

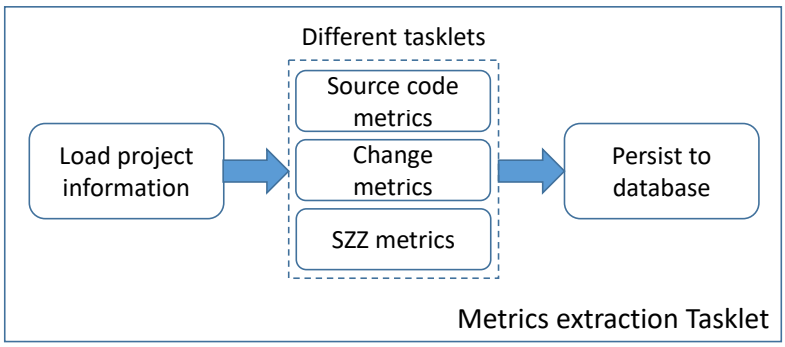


Figure 3.6: Metrics extraction tasklets

3.3.2 Metrics extraction

The extraction and accurate categorization of commits in BICO makes it possible to implement a wide range of software evolution analyses. We have implemented a feature in BICO to extract software metrics of any analyzed system. The supported metrics are the change metrics proposed by Moser *et al.* [10], source code metrics [11], and defect counts from the SZZ algorithm [7]. SZZ can be run once to label Java classes with bug counts across the evolution of the system. Change and source code metrics can be calculated for the system on every commit, on every n^{th} commit, or on a specific commit specified by the commit hash. This functionality allows researchers in the domain of defect prediction to build defect predictors and carry out empirical studies on any git-based system, and not only on the publicly-available datasets such as the bug prediction benchmark [6] or the Tera-PROMISE repository.[4]

For every metric we have a simple tasklet (Figure 3.6) that is managed as a job. These modules are explained in the next section.

3.4 Linking Commits with Issues

BICO requires the user to provide the Git URL and the issue tracker URL of the project to be analyzed. BICO detects issue IDs in commit messages and use them to link commits with issues. This technique is known in the literature and has been used in several studies [7][6] because developers often include in the commit messages the IDs of the issues resolved by the current commit. For instance, one of the commit messages in Apache Flume states:

"FLUME-774. Move HDFS sink into a separate module"

This means that this commit resolves the issue "FLUME-774". BICO uses these IDs to fetch the issues from issue trackers. However, different projects can have different URLs for their issues and the user

[4] `http://openscience.us/repo`

needs to provide the template URL for the projects they need to analyze. Also different issue trackers have different issue IDs. Currently, BICO supports three issue trackers:

1. JIRA[5] is one of the most used issue trackers in the open-source community, provided by Atlassian. Usually, issue reports can be obtained using the following URL template: `https://issues.apache.org/jira/si/jira.issueviews:issue-xml/%s/%s.xml` where the user puts "%s" to instruct BICO that this is the placeholder for the issue ID. When a project uses this tracker, BICO looks in the commit messages for the pattern *WORD-NUMBER* and detects it as an issue ID, as this is the ID format that JIRA gives to issues, *e.g.,* ZOOKEEPER-2688, KAFKA-4744, and TIKA-2261. We have created a regular expression to match JIRA identifiers:

   ```
   (^|\s+|#)\[?(\w+-\d+)\]?(\s|\.|:|;|,)+
   ```

   We tested the accuracy of this matching algorithm with Apache Kafka, one of the biggest Apache Java projects. 2850 identifiers were found. 2800 directly belong to the Kafka project. 2 identifiers are related to the Gradle[6] project. Another 41 are Kafka Improvement Proposals (KIP) that are not listed in the issue tracker but instead in the Apache Wiki[7]. 15 identifiers were matched that do not belong to any project, *e.g.,* UTF-8 or SERVER-1. In the end, we only have 58 false positives out of 2850 matches. This results in a precision of 98%. In the second step of the repository mining, all identifiers that could not be linked to an existing issue in JIRA are thrown away. This ensures that only true issues are stored in the system.

2. Bugzilla[8] is another widely-used issue tracker provided by Mozilla. Similarly to JIRA, the template is usually `https://bugzilla.mozilla.org/show_bug.cgi?ctype=xml&id=%s`. Since issue IDs are just numbers in Bugzilla, for projects using Bugzilla as an issue tracker, BICO looks for any number predecessed by "bug" in the commit message and tries to find an issue with this number as an ID. The used regex is:

   ```
   bug\s(\d+)
   ```

3. GitHub Issues[9] is an issue tracker for the projects hosted on GitHub. More and more projects are moving to GitHub Issues because of its convenience. The user just needs to provide the GitHub URL as the issue tracker, *e.g.,* `https://github.com/elastic/elasticsearch`. Similarly to Bugzilla, GitHub uses plain numbers as issue IDs, sometimes with a number sign in front. The regular expression is very simple and therefore more error-prone than *e.g.,* the one used for JIRA, because the commit message could contain other numbers.

   ```
   \(?#(\d+)\)?
   ```

To mitigate false positives, unlinked issue identifiers are discarded. We cannot prevent that a wrong identifier is detected, but linked to an existing issue. The probability for such a case is fortunately very low.

3.4.1 Issue Analysis

Developers should not blindly trust the issue tracker categorization of issues. As a result, Simon Curty, a student of the *Software Composition Group*, implemented a classifier[10] based on neural networks that

[5] `https://www.atlassian.com/software/jira`
[6] `https://gradle.org`
[7] *e.g.,* `https://cwiki.apache.org/confluence/display/KAFKA/KIP-170`
[8] `https://www.bugzilla.org`
[9] `https://guides.github.com/features/issues`
[10] `https://github.com/curtys/issue-analysis`

Category	Keywords
Bug (corrective)	bug, fix, wrong, error ,fail, problem, patch
Deprecation	deprecation, deprecated
Feature	new, add, requirement, initial, create, feature, implementation
Refactoring	refactor, refactoring
Documentation (non-functional)	docs, documentation, doc
Improvement (perfective)	clean, better, enhancement, improvement, optimization
Merge	merge
Test (preventive)	test, junit, coverage, asset
Dependency upgrade	dependency

Table 3.1: Keywords used for commit message analysis

makes a prediction on the best matching category by processing the information given by the issue tracker entry. We incorporated his work in BICO to have another way to evaluate the category of an issue. In our example project *Kafka* we find many issues, where the issue tracker category does not match the classifier's result. But this could partly also be the result of the classifier's limitations. It cannot differentiate between the same amount of categories as the issue tracker provides. The following categories are supported: bug, improvement, RFE (request for enhancement). Every project in BICO has a link to its commit categorization (project options). This list shows every commit with the associated issues, the issue tracker categorization and the category determined by the classifier. The category based on the commit message is also displayed. This feature is useful for verifying the results and comparing the different categorization methods.

3.4.2 Commit Message Analysis

We implemented a simple heuristic-based categorization mechanism to determine the commit category based on the commit message. Building on the work of Rosen *et al.* [22], we created the list of keywords (see Table 3.1) that are associated with different categories. For every keyword match, the associated category is added to a list. Then the occurrences get counted and the category with the highest keyword occurrences wins.

We are aware of the limitations of such heuristic based methods, but it can give the developer another estimation besides the issue tracker categorization. The case studies in chapter 5 reveal that this approach does not work very well. They give us precisions between 10% and 35%.

3.5 Metrics extraction

We created standalone modules for calculating source code metrics [11], change metrics [10] and defect counts [7] (SZZ algorithm). The SZZ algorithm locates commits of fix-inducing changes. All modules use the Repodriller[11] framework that supports researchers on mining software repositories. The modules are published on GitHub[12].

Each module provides more or less the same functions. There exist filter mechanisms to restrict the data base, *e.g.,* setting the first commit hash, setting a range of commits by start and end date or by commit

[11] `http://repodriller.org`
[12] `https://github.com/papagei9/scg-metrics`

hashes. Additionally, a commit interval can also be specified. Furthermore, setting a list of commits that should be excluded from the analysis is supported. After setting all these parameters, a commit list with all eligible commits is created and ready for further processing.

Source code metrics

We used the proposed metrics from [11], some others provided by Mauricio Aniche and a few determined by us. They are listed in Table 3.3. The module is based on the CK project[13] of Aniche.

Change metrics

The change metrics (Table 3.2) are based on [10] and on Aniche's change metrics project[14].

SZZ algorithm

Automatically identifying commits that induce fixes is an important task, as it enables researchers to quickly and efficiently validate many types of software engineering analyses. Previous work on SZZ, an algorithm designed by Sliwerski *et al.* [7] and improved upon by Kim *et al.* [30] provides a process for automatically identifying the lines modified in a bug-fixing commit, and then identifying the fix-inducing change immediately prior to each line of the bug-fixing commit. SZZ is currently the best available algorithm for this task. With the help of this work we implemented this algorithm as a separate module for BICO.

Name	Description
File	the full file path
Revisions	quantity of commits
Refactorings	quantity of refactorings that occured (if said in commit msg)
Bugfixes	quantity of bugs that file has had (if said in commit msg)
Authors	quantity of different authors
locAdded	total of LOC added
locRemoved	total of LOC removed
maxLocAdded	maximum number of LOC added
maxLocRemoved	maximum number of LOC removed
avgLocAdded	average of LOC added
avgLocRemoved	average of LOC removed
codeChurn	sum of all LOC added and removed
maxChangeset	max number of files committed together with this file
avgChangeset	average number of files committed together
firstCommit	date of the first commit
lastCommit	date of the last commit
weeks	difference in weeks from the last commit - first commit

Table 3.2: Change metrics

[13] https://github.com/mauricioaniche/ck
[14] https://github.com/mauricioaniche/change-metrics

Abb.	Name	Description
CBO	Coupling between objects	Counts the number of dependencies a class has. The tools checks for any type used in the entire class (field declaration, method return types, variable declarations, etc). It ignores dependencies to Java itself (e.g. java.lang.String).
DIT	Depth Inheritance Tree	It counts the number of "fathers" a class has. All classes have DIT at least 1 (everyone inherits java.lang.Object). In order to make it happen, classes must exist in the project (i.e. if a class depends upon X which relies in a jar/dependency file, and X depends upon other classes, DIT is counted as 2).
NOC	Number of Children	Counts the number of children a class has.
NOF	Number of fields	Counts the number of fields in a class, no matter its modifiers.
NOPF	Number of public fields	Counts only the public fields.
NOSF	Number of static fields	Counts only the static fields.
NOM	Number of methods	Counts the number of methods, no matter its modifiers.
NOPM	Number of public methods	Counts only the public methods.
NOSM	Number of static methods	Counts only the static methods.
NOSI	Number of static invocations	Counts the number of invocations to static methods. It can only count the ones that can be resolved by the JDT.
RFC	Response for a Class	Counts the number of unique method invocations in a class. As invocations are resolved via static analysis, this implementation fails when a method has overloads with same number of parameters, but different types.
WMC	Weight Method Class or cyclomatic complexity	It counts the number of branch instructions in a class.
LOC	Lines of code	It counts the lines of count, ignoring empty lines.
LCOM	Lack of Cohesion of Methods	Calculates LCOM metric. This is a not very reliable version.
NOCB*	Number of Catch Blocks	Calculates number of catch blocks.
NONC*	Number of Null Checks	Calculates number of null checks.
NONA*	Number of Null Assignments	Calculates number of null assignments.
NOMWMOP*	Number of Methods with more than one Parameter	Calculates the number of methods with more than one parameter.

Table 3.3: Source code metrics

* were added by us to fulfill research requirements.

3.6 Big Commit Detection

BICO implements an outlier detection method to find big commits, as in Figure 4.5(3). Since BICO keeps data about the number of changed files and the number of changed lines in each commit, it can calculate the first quartile Q1 and the third quartile Q3 of these metrics in each category of commits. Using these quartiles, BICO uses the definition of *extreme outliers* in statistics to detect big commits. Such outliers are data points that are smaller than $Q1 - 3 \times IQR$ (*i.e.,* lower outer fence) or larger than $Q3 + 3 \times IQR$ (*i.e.,* upper outer fence), where $IQR = Q3 - Q1$ (intermediate quartile range). In BICO, big commits are commits that have more changed files or lines of code than the extreme outlier threshold within that commit category. The main rationale behind this approach is that different types of source code changes have different characteristics. For instance, code refactoring changes tend to be large whereas bug-fix changes are known to be small [12].

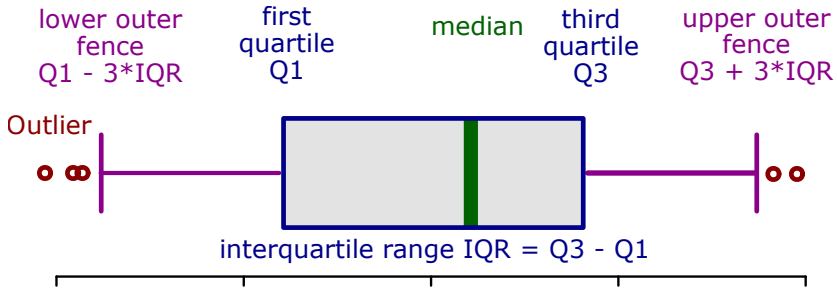


Figure 3.7: Statistical outlier detection

3.7 Test Repository

We have created a git repository for a synthesized Java project named *AcmeStore*[15]. It simulates a simple article store for CDs, DVDs and Books with customers. This repository contains commits of different categories connected to GitHub issues and acts as the ground truth to test BICO. We are currently in the process of expanding this repository to cover more cases and scenarios to be able to test BICO more extensively in the future. The metrics of the ground truth have been manually measured and can be found on `https://github.com/papagei9/AcmeStore/blob/master/README.md`.

[15] `https://github.com/papagei9/AcmeStore`

4
Features of BiCo

All extracted data is saved in a database and can be used separately from BICO itself. However, BICO provides its own UI for users to control the software and explore the extracted data. The UI is explained step by step and illustrated with examples of analyzing Apache Kafka[1] in the following subsections.

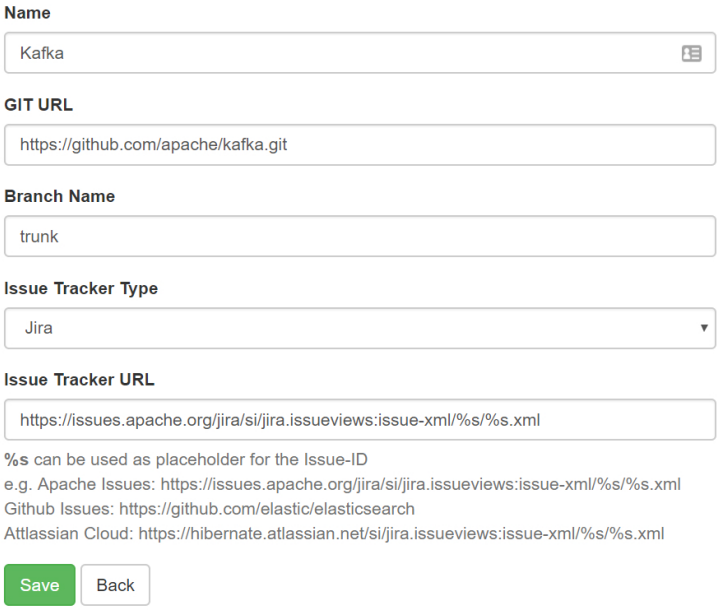


Figure 4.1: Adding projects to BICO

[1] `https://kafka.apache.org`

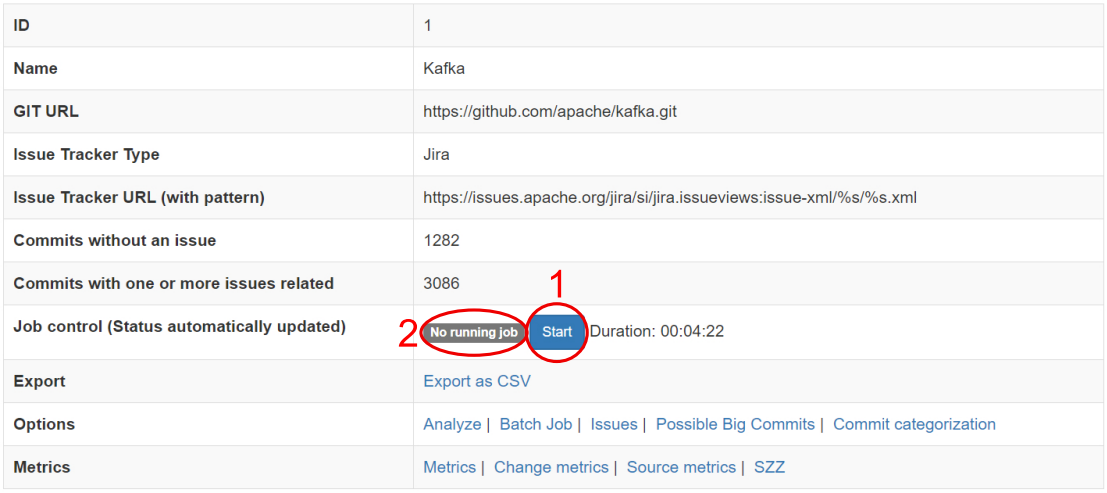


Figure 4.2: Screenshot of the project detail site

4.1 Managing and Processing Projects

To add projects, only very little information is needed as shown in Figure 4.1. Just enter a name, the Git HTTPS URL, the name of the targeted branch, the issue tracker type and the URL to the issue tracker. BICO does currently not support SSH access to Git repositories. The tool already shows some issue tracker examples. These details can be changed later in the project view. Deleting a project may need some minutes if jobs were already executed and data was saved.

Now clicking on "Start" (Figure 4.2 (1)) will run the background batch process for parsing the repository and linking the issues to the specified issue tracker. This will need some time in relation to the repository size. The job could also be managed through the Spring Batch Admin[2] interface, linked as "Batch Job" in the project details. As soon as the job is finished, the user gets an indication (Figure 4.2 (2)) and the commit list as shown in Figure 4.3 will appear and the commits can be browsed.

The commit view (Figure 4.4) contains some information about the commit itself, the linked issue(s) and a list of changed files. For every file, a highlighted diff is shown to illustrate the changes made in that specific file.

4.2 Project Analytics

The analytics include a list of possible big commits seen in Figure 4.5 and some statistics about the issue categories. Big commits are found by an outlier detection explained in the section 3.6 Big Commit Detection.

4.2.1 Big commits

Big commits are currently separated into two categories: MERGE and OUTLIER. Commits labelled as MERGE are just merge commits that are very big. OUTLIER commits are detected by our outlier detection.

[2] `http://docs.spring.io/spring-batch-admin`

Commits (4368)

ID	Issues	Additions	Deletions	FirstLineOfMessage
1	0	72944	0	Initial checkin of Kafka to Apache SVN. This corresponds to https://github.com/kafka-dev/kafka/commit/709afe4ec75489bc00a44335de8821fa726bb97e except that git specific files have been removed and code has been put into trunk/branches/site/etc. This is just a copy of master, branches and history are not being converted since we can't find a good tool for it.
2	0	14	0	gitignore for git-svn users.
3	1	0	0	upgrade zkclient jar; #KAFKA-82
4	1	33	31	commit offset before consumer shutdown KAFKA-84; rename util.StringSerializer to ZKStringSerializer to avoid confusion with producer.StringSerializer
5	1	4	13	Options in SyncProducerConfig and AsyncProducerConfig can leak, KAFKA-83
6	1	2	0	change default zk connection limit to infinite; KAFKA-88
7	0	5	4	Update readme links.
8	1	1380	1088	KAFKA-93 | Change and add ASF source header to follow standard ASF source header (http://www.apache.org/legal/src-headers.html).
9	1	74	46	KAFKA-93 | add license to missed files and remove LinkedIn copyright line per ASF guideline. Thanks to Joel Koshy for pointing it out
10	1	712	151	auto-discovery of topics for mirroring; patched by Joel; reviewed by Jun; KAFKA-74

Figure 4.3: Screenshot of the project commit list

ID	220
Name	single_host_multi_brokers system test fails on laptop; patched by John Fung; reviewed by Jun Rao; kafka-413
Changes	4
Additions	2
Deletions	2
Files changed	1
Date	26-07-2012 18:59
Ref	2e073db8dc3ce51c993f2
Number of issues assigned to this commit	1
Parent commit	219: KAFKA-405 Improve .highwatermark file; patch
Commit category based on commit message	Test

Message

single_host_multi_brokers system test fails on laptop; pat

git-svn-id: https://svn.apache.org/repos/asf/incubator/kafk

File list

ID	ChangeType	Path	Additions	Deletions	Changes
2834	MODIFY	system_test/single_host_multi_brokers/bin/run-test.sh	2	2	4

Files changed (1) show

system_test/single_host_multi_brokers/bin/run-test.sh CHANGED

```
@@ -29,7 +29,7 @@
29 29  source   ${common_dir}/util.sh                       # include the util script
30 30
31 31  readonly base_dir=$(dirname $0)/..                   # root of this test suite
32    - readonly base_dir_full_path=$(readlink -f $base_dir)  # full path of the root of this test suite
   32 + readonly base_dir_full_path=`cd $base_dir; pwd`      # full path of the root of this test suite
33 33  readonly config_dir=${base_dir}/config
34 34
35 35  readonly test_start_time="$(date +%s)"               # time starting the test
@@ -264,7 +264,7 @@
264 264          first_data_file_dir=${kafka_data_log_dirs[$i]}/${test_topic}-0
265 265          first_data_file=`ls ${first_data_file_dir} | head -1`
266 266          first_data_file_pathname=${first_data_file_dir}/$first_data_file
267    -        kafka_first_data_file_sizes[$i]=`stat -c%s ${first_data_file_pathname}`
   267 +        kafka_first_data_file_sizes[$i]=`ls -l ${first_data_file_pathname} | awk '{print $5}'`
268 268          kafka_first_data_file_checksums[$i]=`cksum ${first_data_file_pathname} | awk '{print $1}'`
269 269          info "## broker[$i] data file: ${first_data_file_pathname} : [${kafka_first_data_file_sizes[$i]}]"
270 270          info "##      ==> crc ${kafka_first_data_file_checksums[$i]}"
```

Issue list

ID	Name	Type	Classifier Type	Priority
158	KAFKA-413	Bug	BUG	Major

Figure 4.4: Screenshot of the commit detail view

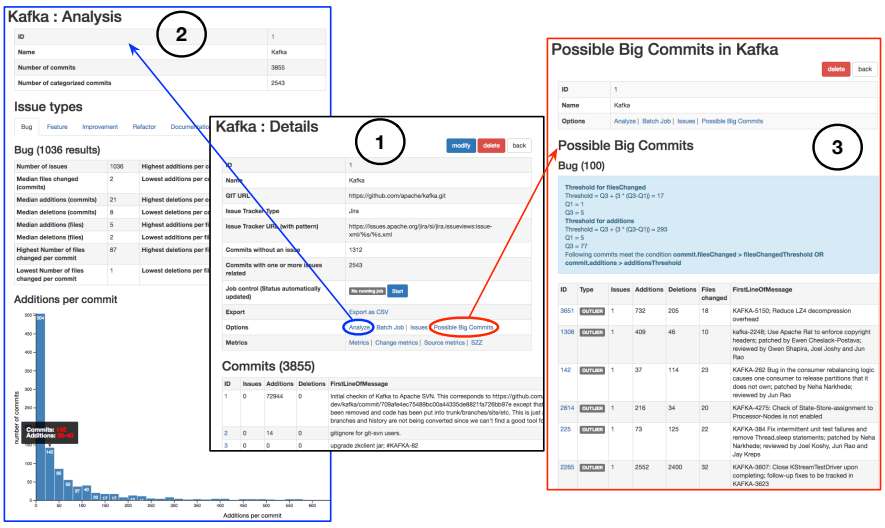


Figure 4.5: Screenshots of BICO showing the main functionality. From the overview of project details (1), one can navigate to an analysis of the project (2), or to a list of possible big commits (3).

4.2.2 Statistics

The statistics show us "files changed per commit" and "additions per commit" graphs for the issue categories. *e.g.*, in the Apache Kafka project we clearly find outliers just by looking at the grahps in Figure 4.6 and Figure 4.7. Defining a sound threshold could expose a part of the commits as not belonging to that category.

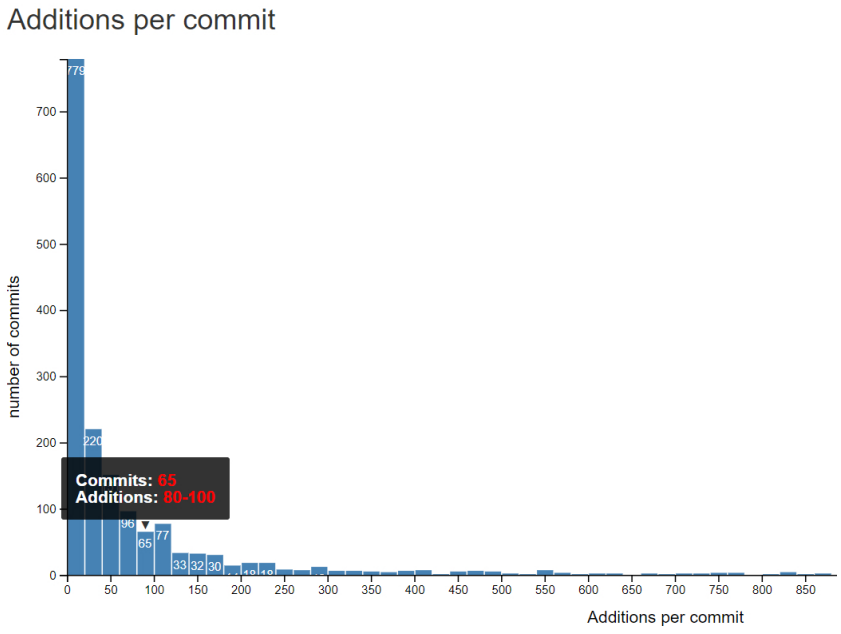


Figure 4.6: Apache Kafka: additions per commit in the Bug category

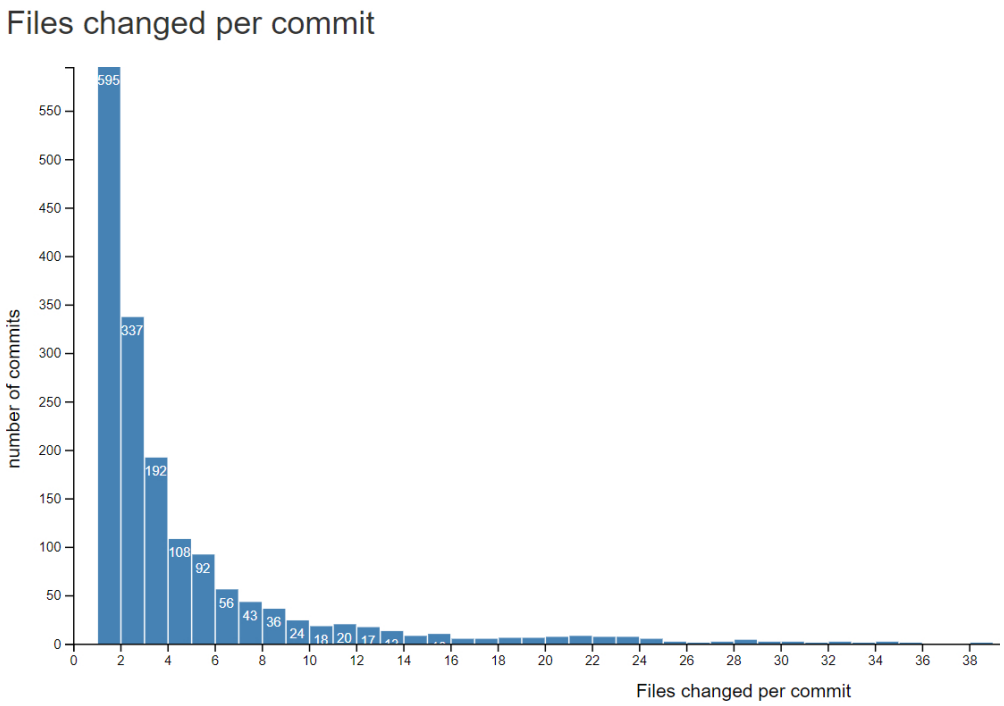


Figure 4.7: Apache Kafka: files changed per commit in the Bug category

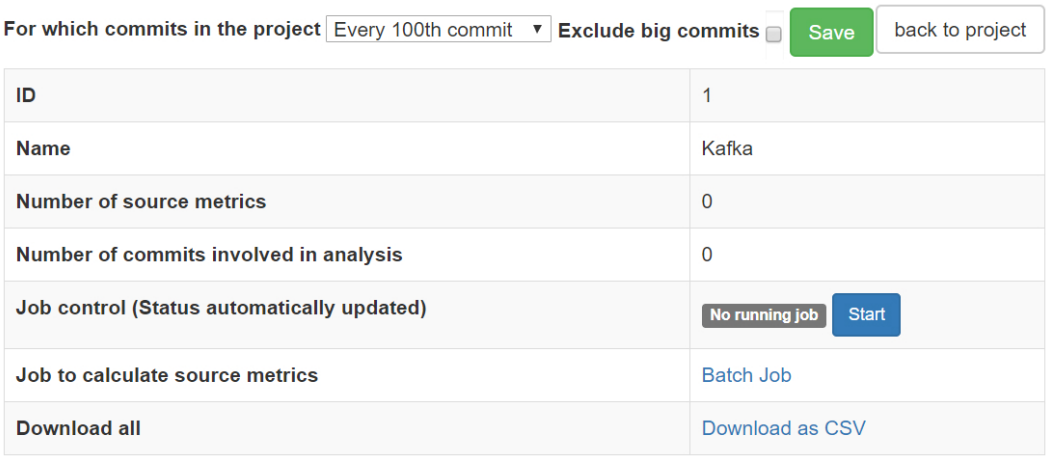


Figure 4.8: Source code metrics interface of Apache Kafka

4.3 Metric Suite Extractor

Metrics can always be exported as a CSV file to use for further research. One can decide if the big commits should automatically be excluded from the metrics calculation. The procedure is the same for every type of metric.

Figure 4.9: Extract metrics of a specific commit

4.3.1 Single commit metric extraction

To get the metrics of just a single commit, a separate form like in Figure 4.9 exists that takes a commit ref and the sliding window for the change metrics. Big commits from the analysis could be excluded for more accurate results. It then calculates the metrics and does not take them from the database, since they could be missing. After the calculations are finished, the results are delivered as a CSV file.

4.3.2 Source code metrics

The tool lets us extract a list of source metrics. The interface shown in Figure 4.8 provides us with two simple options: for every n^{th} commit the metrics should be generated and if big commits (from the analysis) should be excluded. After the first run of the job is done, we get a list ordered by time (Figure 4.10) of all points where metrics were generated. If we click on a ID in this list, the tool will show us the generated metrics for every file (Figure 4.11).

4.3.3 Change metrics

Further, the tool is able to generate change metrics of the project files. Again, the same settings as for source code metrics can be applied. Additionally, a sliding window (Figure 4.12) is defined. At every commit, source metrics are generated within this time window (from every commit backwards in history). After running the job, the time points (in the meaning of commits) are displayed as a ordered list (Figure 4.13). Clicking on an entry ID leads us to the specific change metrics of each file (Figure 4.14).

4.3.4 SZZ algorithm

The SZZ algorithm [7], or also called *defect counts* has a simpler interface (Figure 4.15). Big commits can still be excluded if desired. The results are displayed as a file list (Figure 4.16) with commit and bugfix count. Clicking on a specific file displays a commit list with a bug count shown in Figure 4.17). Further, each commit that we categorized as bugfix is labelled as *BUGFIX*.

Source metrics commits

ID	Commit date	FirstLineOfMessage	# metrics
3668	01-06-2017 04:30	KAFKA-5350: Modify unstable annotations in Streams API	1290
3768	13-06-2017 14:33	KAFKA-5418; ZkUtils.getAllPartitions() may fail if a topic is marked for deletion	1302
3868	30-06-2017 07:47	KAFKA-5544; The LastStableOffsetLag metric should be removed when partition is deleted	1310
3968	28-07-2017 20:28	KAFKA-5341; Add UnderMinIsrPartitionCount and per-partition UnderMinIsr metrics (KIP-164)	1331
4068	27-08-2017 01:33	MINOR: Fix doc typos and grammar	1417
4168	18-09-2017 14:22	KAFKA-5893: more logging for investigating ResetIntegrationTest failures	1493
4268	04-10-2017 19:57	KAFKA-5856; AdminClient.createPartitions() follow up	1524
4368	27-10-2017 17:40	MINOR: Remove TLS renegotiation code	1529

Figure 4.10: Incomplete list of source code metrics snapshots of Apache Kafka

Source metrics

File	CBO	WMC	DIT	NOC	RFC	LCOM	NOM	NOPM	NOSM	NOF	NOPF	NOSF	NOSI	LOC
security/scram/ScramSaslServerProvider.java	class	2	3	5	0	5	1	2	1	1	1	0	1	2
common/requests/RequestHeader.java	class	8	29	2	0	17	0	13	12	2	10	1	6	5
common/record/DefaultRecordBatch.java	class	18	130	4	0	83	1438	63	47	7	39	1	32	15
rest/entities/ConfigKeyInfo.java	class	3	19	1	0	8	5	15	15	0	11	0	0	12
common/record/MutableRecordBatch.java	interface	3	4	1	0	0	6	4	0	0	0	0	0	0
processor/internals/Checkpointable.java	interface	1	2	1	0	0	1	2	0	0	0	0	0	0
connect/errors/SchemaBuilderException.java	class	1	3	7	0	0	3	3	3	0	0	0	0	0
kafka/clients/ClientResponse.java	class	4	13	1	0	1	21	11	11	0	8	0	0	0

Figure 4.11: Incomplete list of source code metrics of at a specific time of Apache Kafka

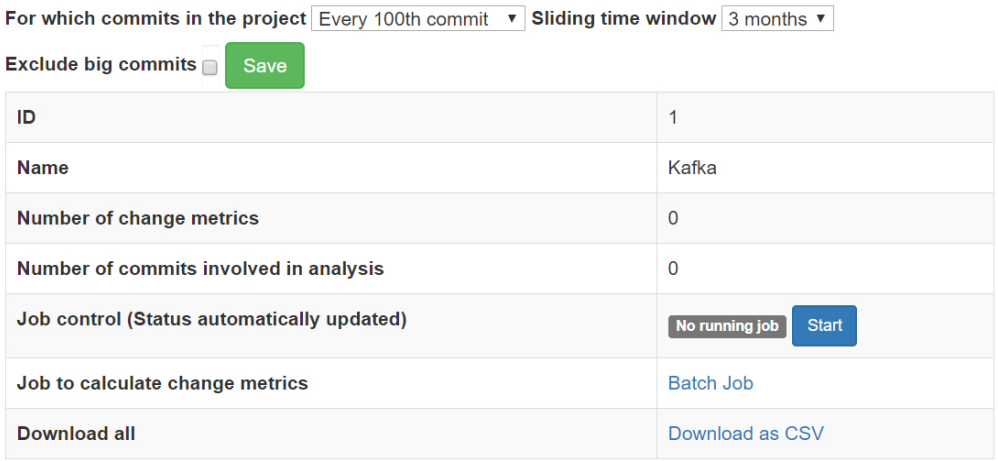


Figure 4.12: Change metrics interface of Apache Kafka

Change metrics commits (5)

ID	Commit date	FirstLineOfMessage	Amount of metrics
329	23-10-2012 03:26	KAFKA-541 Move to metrics csv reporter for system tests; patched by Yang Ye; reviewed by Neha, Jun and Joel	6
1368	06-08-2015 01:33	MINOR: auto.offset.reset docs not in sync with validation	102
2368	13-05-2016 03:15	KAFKA-3421: Connect developer guide update and several fixes	466
3368	30-03-2017 20:24	MINOR: Update possible errors in OffsetFetchResponse	1145
4368	27-10-2017 17:40	MINOR: Remove TLS renegotiation code	769

Figure 4.13: Incomplete list of change metrics snapshots of Apache Kafka

Change metrics

File	Revisions	Refactorings	Bugfixes	Authors	LOC+	LOC-	avgLOC+	avgLOC-	maxLOC+
kstream/internals/KStreamKStreamJoin.java	1	0	0	1	1	0	1.00	0	1
common/network/Send.java	1	0	0	1	2	2	2.00	2.00	2
kafka/test/ProcessorTopologyTestDriver.java	8	1	3	6	48	10	6.00	1.25	21
kafka/clients/ClusterConnectionStatesTest.java	2	0	1	2	201	0	100.50	0	181
kstream/internals/ProducedInternal.java	1	0	0	1	39	0	39.00	0	39
clients/admin/DescribeLogDirsResult.java	2	0	0	2	71	1	35.50	0.50	70
security/scram/ScramSaslServerProvider.java	1	0	0	1	2	1	2.00	1.00	2
common/requests/RequestHeader.java	3	2	0	2	61	19	20.33	6.33	33

Figure 4.14: Incomplete list of change metrics of at a specific time of Apache Kafka

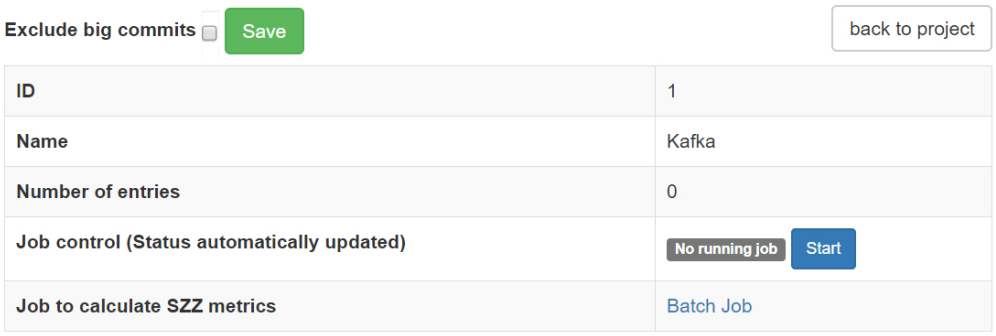
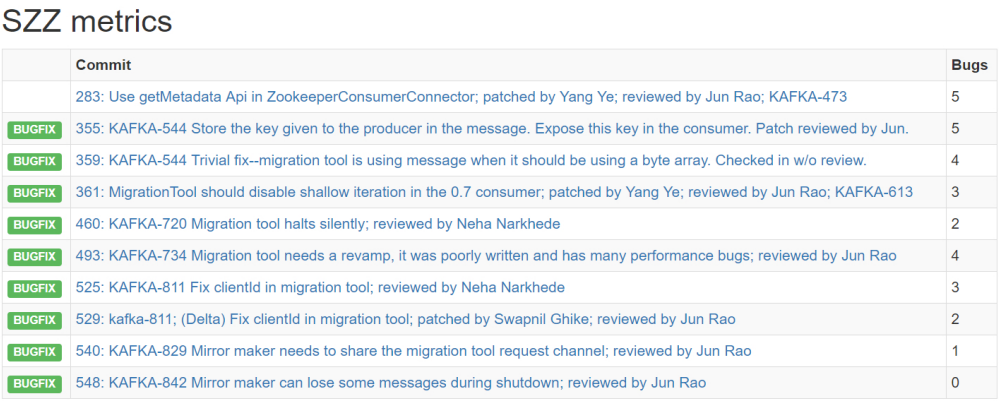


Figure 4.15: SZZ algorithm interface of Apache Kafka

SZZ files (1667)

File	Commits	Bugfixes
clients/src/main/java/org/apache/kafka/clients/ClientRequest.java	13	0
clients/src/main/java/org/apache/kafka/clients/ClientUtils.java	1821	6
clients/src/main/java/org/apache/kafka/clients/ClusterConnectionStates.java	3415	6
clients/src/main/java/org/apache/kafka/clients/CommonClientConfigs.java	2554	2
clients/src/main/java/org/apache/kafka/clients/ConnectionState.java	6	0
clients/src/main/java/org/apache/kafka/clients/InFlightRequests.java	1983	2
clients/src/main/java/org/apache/kafka/clients/KafkaClient.java	3038	5
clients/src/main/java/org/apache/kafka/clients/Metadata.java	3315	17

Figure 4.16: Incomplete list of files analyzed by the SZZ algorithm in Apache Kafka



SZZ metrics

	Commit	Bugs
	283: Use getMetadata Api in ZookeeperConsumerConnector; patched by Yang Ye; reviewed by Jun Rao; KAFKA-473	5
BUGFIX	355: KAFKA-544 Store the key given to the producer in the message. Expose this key in the consumer. Patch reviewed by Jun.	5
BUGFIX	359: KAFKA-544 Trivial fix--migration tool is using message when it should be using a byte array. Checked in w/o review.	4
BUGFIX	361: MigrationTool should disable shallow iteration in the 0.7 consumer; patched by Yang Ye; reviewed by Jun Rao; KAFKA-613	3
BUGFIX	460: KAFKA-720 Migration tool halts silently; reviewed by Neha Narkhede	2
BUGFIX	493: KAFKA-734 Migration tool needs a revamp, it was poorly written and has many performance bugs; reviewed by Jun Rao	4
BUGFIX	525: KAFKA-811 Fix clientId in migration tool; reviewed by Neha Narkhede	3
BUGFIX	529: kafka-811; (Delta) Fix clientId in migration tool; patched by Swapnil Ghike; reviewed by Jun Rao	2
BUGFIX	540: KAFKA-829 Mirror maker needs to share the migration tool request channel; reviewed by Jun Rao	1
BUGFIX	548: KAFKA-842 Mirror maker can lose some messages during shutdown; reviewed by Jun Rao	0

Figure 4.17: Incomplete list of defect counts of a specific file in Apache Kafka

5 Case Studies

5.1 Apache Kafka Case Study

As an initial evaluation, we use BICO to extract and analyze the bug-fix commits in Apache Kafka (4400 commits), a popular publish/subscribe distributed infrastructure implemented on top of Hadoop.[1] Kafka uses GitHub as the source code management system and *JIRA* as an issue tracker. BICO was able to categorize 1036 commits as fixes using the categories of the linked issues. However, BICO detects that 100 of the fixing commits (9.6%) are big commits. We manually investigated the reported big commits in the bug-fix category and observe that only 15 of them are false positives, *i.e.,* they are actually bug-fix commits. Some have new tests added that confuse the outlier detection, some commits provide fixes for concurrency problems involving many files. The remaining 85 commits are correctly classified as big commits: 33 improvement, 19 multipurpose, 15 feature addition, 9 refactoring, 8 test addition, and 2 documentation addition. This precision of 85% suggests that the statistical outlier detection is a reliable method for detecting big commits and BICO can be used off-the-shelf to analyze project repositories and aid researchers in related empirical studies.

5.2 Apache Flume Case Study

We also analyze Apache Flume (1730 commits), which is a distributed, reliable, and available system for efficiently collecting, aggregating and moving large amounts of log data from many different sources to a centralized data store.[2] We use BICO to extract and analyze bug-fix commits. GitHub is used as source code management system and *JIRA* as issue tracker. BICO categorized 743 commits as bug fixes using the categories of the linked issues. 9.6% of the commits (73 out of 761) are detected as big commits by the statistical outlier detection. While manually investigating the big commits and the related changes, we re-classified them. This showed us that only 24 are actual bug fix commits. A few commits fix concurrency problems involving many files. Also many commits have new tests added that blew up the size and therefore were detected as outlier. The other 49 commits are correctly classified as big commits: 16

[1] `https://github.com/apache/kafka`
[2] `https://github.com/apache/flume`

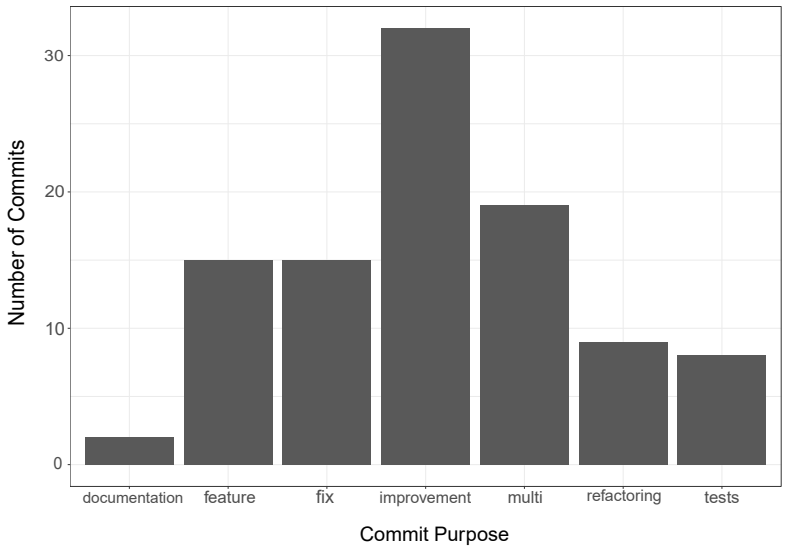


Figure 5.1: Bug-fix big commits investigation of Apache Kafka

improvement, 12 multipurpose, 9 feature addition, 9 non-functional changes, 3 test addition. This results in a correct detection of big commits in 67.1% of all cases. With the help of the mentioned investigation, we also examined the accuracy of the issue classifier (subsection 3.4.1) in the case of bug-fix big commits. Around 65.8% (48 out of 73) of the big commits were correctly classified by the issue classifier as big commits. Our simple heuristic-based algorithm to categorize a commit by its commit message has an accuracy of 35.6% (26 out of 73 were categorized correctly) in this context. 39 commits could not be categorized at all and 8 commits did have a wrong categorization.

5.3 Apache Storm Case Study

To also include a bigger Java project, we took Apache Storm[3], a distributed realtime computation system, with approximately 9000 commits. Over 1300 commits have been categorized as Bug, Feature, *etc.* with the help of the issue tracker, and over 390 big commits were found by the analysis (see Table 5.1). 7521 commits are uncategorized due to missing issue tracker identifiers. We analyze the detected big commits to know how accurate the results are. This includes manual categorization of these big commits. We examine the precision of the outlier and merge detection (in Table 5.3), and also compare the three used issue categorization mechanisms: issue tracker, machine learning issue classifier and commit message-based heuristic approach (in Table 5.3). We split the measurement of the categorization accuracy in two parts: full match and partial hit. A full match means that the issue category and the manually determined category are equal. For multi-purpose commits, we can never have a full match since all our automatic categorization mechanisms only support one category. For partial hits, which means that at least one of the commit categories must be covered by the issue.

Only the categories Feature, Documentation, Bug and Improvement are taken in account for this case study. Since the machine learning classifier uses the category *RFE* (Request for enhancement) in the meaning of *Feature*, we treat them as equivalent.

[3] `https://github.com/apache/storm`

Our methodology for investigating the data was to first check the different categorization results for each big commit in the selected categories. Then read the commit messages and check the file changes. As a last validation source, we used the Jira issues and the linked GitHub pull requests to determine the best category, or in case of multi-purpose the best categories.

We now discuss the results and check if the categorization is really correct. This initial categorization by BICO is based on the issue type given by the issue tracker.

Category	**Commits**	**Big commits**
Feature	148	38
Documentation	42	16
Bug	679	198
Improvement	364	120
Subtask	34	*
Refactor	10	*
Wish	2	*
Test	4	*
Task	15	*
Dependency upgrade	16	*
Other	14	*
N/A	50	20
Uncategorized	7521	-
Total	8899	392

Table 5.1: Overview over categories and commits in Apache Storm project from initial categorization by the issue tracker
* These categories are not included in the big commit analysis.

Category	**Outliers**	**Outlier precision**	**Merge**	**Merge precision**	**Multi-purpose**	**Miscategorized**
Feature	16	50.00%	22	59.09%	55.26%	44.74%
Documentation	10	0.00%	6	0.00%	0.00%	100%
Bug	131	36.64%	67	34.33%	35.86%	64.14%
Improvement	79	51.90%	41	68.29%	57.50%	42.50%

Table 5.2: Results of the investigation of Apache Storm's big commit analysis

Category	**Issue tracker**	**Issue classifier**	**Commit message**	
Feature	97.37%	94.74%	21.05%	accuracy partial hit
	42.11%	44.74%	5.26%	accuracy full match
Documentation	100%	0.00%	31.25%	accuracy partial hit
	100%	0.00%	31.25%	accuracy full match
Bug	57.07%	45.96%	23.23%	accuracy partial hit
	35.35%	28.79%	13.64%	accuracy full match
Improvement	50.83%	53.33%	19.17%	accuracy partial hit
	17.50%	20.00%	5.83%	accuracy full match

Table 5.3: Categorization of big commits

Feature category

In the feature category we have a total of 148 commits, where 38 are marked as big commits. These 38 big commits are examined further. The manual investigation of the outlier detection shows us that 50%

of these outliers are actually big commits *i.e.,* multi-purpose commits. For *merge* labelled big commits, 59.09% are correct. All these big commits typically consists of feature addition, documentation and tests. 44.74% of the as big commit marked commits in this category are actually single-purpose and should not be listed as big commit. A feature commit can have different appearances. For example, only very few files touched, but many lines of codes changed. The opposite with many files touched, but only a few lines of code changed was also observed. This means there is a big discrepancy between feature commits. Good developers write tests for newly implemented features and improvements. We noticed that this does not happen every time and therefore a commit including new tests is considered as multi-purpose.

Now, we take a look at the different categorization approaches. The issue categorization has a precision of 42.11% for full matches. For partial hits we get a precision of 97.37%. The machine learning classifier performed similar to the issue categorization. 44.74% of the big commits in this category were correctly full matched. A 94.74% accuracy is achieved for partial hits. The commit message categorization gives a very low precision of 5.26% for full matches and 21.05% for partial hits. This shows us that a simple heuristic based approach cannot deliver accurate results in this category. We assumed that it is almost impossible to categorize merge commits by this method, since the commit message often just contains "merge branch X of Y into Z".

Documentation category

This category contains only 42 commits, whereas 16 commits are detected as big commits. The investigation of these big commits shows us that no one actually is multi-purpose but just documentation stuff. Because of this, the precision of the outlier detection is 0%. 6 of these 16 big commits are *merge* labelled commits. This gives us a precision for merge big commits of also 0%. In the end, all big commits in this category are false positives.

Since the initial categorization is based on the issue tracker, the precisions of the issue tracker full match and partial hit are both 100%. On the other side, the machine learning classifier failed to predict the correct category in every single case. The heuristic based approach has a precision of 31.25% over the 16 big commits.

Bug category

The bug category contains 679 commits, where 198 are detected as big commits. Going through these 198 big commits, we see that 131 are marked as outlier and 67 are merge commits. After manual categorization of these 198 commits, we get a precision of 36.64% for the outlier detection, and 34.33% for the merge commits. 35.86% of the investigated commits are really multi-purpose, *i.e.,* big commits and 64.14% are miscategorized. This high false positive rate can be explained by the heterogeneity of this category. Many big commits in this category actually are not bug fixes but something other, *e.g.,* feature addition. Because of this diversity, the calculated threshold values for the outlier detection is not adapted to bug-fix commits but instead to a whole bunch of different types of commits. Due to this, we find that wrongly categorized uni-purpose commits appear to be marked as outliers more often than correctly categorized uni-purpose commits.

We noticed that many bug-fix commits contain new tests that test the fixed functionality. We considered such commits as uni-purpose. Also, when a bug is fixed and an existing test is only adapted to the fixed functionality, it is considered as uni-purpose, too.

For full matches, the issue tracker has a precision of 35.35% and 57.07% for partial hits. The precision of the machine learning approach is 28.79% for full matches and 45.96% for partial hits. The commit message approach gives us precisions of 13.64% for full matches and 23.23% for partial hits.

Improvement category

In the improvement category are 364 commits, where 120 actually are marked as big commits. The manual categorization was quite difficult, because deciding between feature and improvement was not an easy task. This means, this category is predestined to have many false positives. Regarding the amount of changes in a commit gives us a good estimation if it is too big for just being an improvement and rather should be a feature addition.

The issue tracker's precision for full matches is 17.50% and 50.83% for partial hits. The issue classifier has a precision of 20.00% for full matches and 53.33% for partial hits. The commit message-based approach has a low precision of 5.83% for full matches and 19.17% for partial hits.

Conclusions

We are aware that we could not define the whole set of relevant (big) commits for each category, because analyzing 9000 commits by hand is a bit out of scope in this context. Because of this, we never know if there are any big commits that were not detected and therefore we cannot make any statement about the recall values. However, in a sample of 1% of normal commits, just 2 multi-purpose commits were found. The Apache Storm project has many merge commits in comparison with other projects like Apache Kafka or Apache Flume. We see that the categorization results are not that accurate for big commits. This mostly comes from the way developers and users create Jira issues for this project. Many bug reports are not actually bugs but more feature and improvement requests. Also, in many cases the reporters fail to correctly distinguish between improvement and feature. This supports our finding that the boundaries between feature and improvement commits are blurry. We should probably treat these two categories as one. Merging those categories results in a outlier precision of 51.58% and a merge precision of 65.08% which is not significantly different from the single values of each category.

Measurements

Category	Feature	Documentation	Bug	Improvement
Total commits	38	16	198	120
Total merge commits	22	6	67	41
Total non-merge commits	16	10	131	79
Issue full match	16	16	70	21
Issue full match precision	42.11%	100%	35.35%	17.50%
Issue partial hit	37	16	113	61
Issue partial hit precision	97.37%	100%	57.07%	50.83%
Issue non-merge correct	7	10	41	14
Issue non-merge incorrect	9	0	90	65
Issue non-merge precision	43.75%	100%	31.30%	17.72%
Issue merge correct	9	6	29	7
Issue merge incorrect	13	0	38	34
Issue merge correct precision	40.91%	100%	43.28%	17.07%
Classifier full match	17	0	57	24
Classifier full match precision	44.74%	0%	28.79%	20.00%
Classifier partial hit	36	0	91	64
Classifier partial hit precision	94.74%	0%	46.96%	53.33%
Classifier non-merge correct	8	0	36	18
Classifier non-merge incorrect	8	0	95	61
Classifier non-merge precision	50%	0%	27.48%	22.78%
Classifier merge correct	9	0	21	6
Classifier merge incorrect	13	6	46	35
Classifier merge correct precision	40.91%	0.00%	31.34%	14.63%
Commit full match	2	5	27	7
Commit full match precision	5.26%	31.25%	13.64%	5.83%
Commit partial hit	8	5	46	23
Commit partial hit precision	21.05%	31.25%	23.23%	19.17%
Commit non-merge correct	2	5	27	7
Commit non-merge incorrect	14	5	104	72
Commit non-merge precision	12.50%	50%	20.61%	8.86%
Commit merge correct	0	0	0	0
Commit merge incorrect	22	6	67	41
Commit merge correct precision	0.00%	0.00%	0.00%	0.00%
Multi-purpose commits	21	0	71	69
Multi-purpose non-merge commits	8	0	48	41
Multi-purpose merge commits	13	0	23	28
Multi-purpose of non-merge commits rate	50.00%	0.00%	36.64%	51.90%
Multi-purpose of merge commits rate	59.09%	0.00%	34.33%	68.29%

Table 5.4: All results of the investigation of Apache Storm's big commit analysis

6 Conclusions and Future Work

Analyzing software evolution often requires the purposes of code changes to be determined. BICO, built with Java Spring, is a tool that links information from software code management systems and issue trackers to determine the purposes of source code changes, *i.e.,* commits. BICO can be used off-the-shelf to analyze software projects, build datasets on software changes, extract software metrics for defect prediction, and explore the collected data. Further, the tool helps diagnosing big commits and provides simple statistical analytics. The experimental results on real-world systems also demonstrate the usefulness of BICO. The tool provides a simple user interface to manage projects and navigate through commits and issues. The data mining and metrics extraction are realized by batch processing for reliable and fast information gathering. This leads us to future work and limitations.

BICO could support more software repositories. The tool currently does not support more issue trackers like Mantis[1] or Redmine[2]. Also, the mining of Subversion[3] repositories is not implemented.

The tool could be improved to store all metadata about commits and issues that can be retrieved, so developers will not be restricted if they need more information than we currently provide. For example, the commit author could be used to get some statistics about a specific developer (how many commits, lines changed, etc.). Issue trackers provide a detailed issue description and in many cases also patch files. Issues could link to duplicates and dates of when the issue was created and resolved are also accessible. There are plenty of other properties that could be used for statistics and improving the user experience.

The entire tool is based on the assumption that developers include issue IDs in commit messages to link their commits with the issues. There exist several projects whose developers do not apply this common known rule. This was also confirmed in the studies by Bird *et al.* [27] and Bachmann *et al.* [26], who reported that almost 54% of fixed bugs in bug repositories are not linked to commit documentation. Researchers have also proposed more complex methodologies to link issues to commits. The approaches of Wu *et al.* [28] go beyond the usage of simple string matching algorithms. Such approaches were not in the scope of this work, but can be used as inspiration for further extensions and improvements.

We saw in the case studies that a multi category classifier that gives us a list of categories with probabilities for a specific commit would be very useful, because the current sources (issue tracker, commit

[1] `http://www.mantisbt.org`
[2] `http://www.redmine.org`
[3] `https://subversion.apache.org`

message, machine learning classifier) only propose one category. Additionally, for big commits, the commit message based approach is not giving us any good results and therefore should be improved or removed. Currently we only rely on outlier and merge detection for the big commit analysis. Having other detection mechanisms could improve the analysis significantly.

It would be nice for developers, if the tool was publicly ready-to-use available as a web service. At this moment, users have to deploy the tool by themselves. But we also provide a Virtual Machine with BICO already set up. This would require the implementation of an authentication management and some other improvements in stability and reliability.

Programmatic access through a set of APIs to the features of BICO would be a very useful enhancement helping developers to directly implement our work in their programs without using the web interface.

Acknowledgements

I would like to thank Haidar Osman for guiding me through this project, and supporting me to continue my seminar work and creating a nice software evolution research tool. I thank Oscar Nierstrasz for his patience and trust, and giving me the opportunity to do another thesis at the Software Composition Group. Finally, I appreciate the work of Simon Curty for providing an API to his issue classifier, so I could add more functionality to BICO.

Bibliography

[1] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, no. 3, pp. 286–315, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9077-5

[2] H. Osman, M. Lungu, and O. Nierstrasz, "Mining frequent bug-fix code changes," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, Feb. 2014, pp. 343–347. [Online]. Available: http://scg.unibe.ch/archive/papers/Osma14aMiningBugFixChanges.pdf

[3] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 213–224.

[4] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, "Predicting source code changes by mining change history," *Software Engineering, IEEE Transactions on*, vol. 30, no. 9, pp. 574 – 586, Sep. 2004.

[5] E. Giger, M. Pinzger, and H. C. Gall, "Can we predict types of code changes? an empirical analysis," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 2012, pp. 217–226.

[6] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*. IEEE CS Press, 2010, pp. 31–40.

[7] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of International Workshop on Mining Software Repositories — MSR'05*. Saint Lous, Missouri, USA: ACM Press, 2005.

[8] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*. ACM, 2008, p. 23.

[9] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 392–401.

[10] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 181–190. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368114

[11] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994. [Online]. Available: http://dx.doi.org/10.1109/32.295895

[12] Lucia, F. Thung, D. Lo, and L. Jiang, "Are faults localizable?" in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, Jun. 2012, pp. 74–77.

[13] A. Mockus and L. Votta, "Identifying reasons for software change using historic databases," in *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*. IEEE Computer Society Press, 2000, pp. 120–130.

[14] A. K. Midha, "Software configuration management for the 21st century," *Bell Labs Technical Journal*, vol. 2, no. 1, pp. 154–165, 1997.

[15] R. Purushothaman and D. E. Perry, "Towards understanding the rhetoric of small changes-extended abstract," in *International Workshop on Mining Software Repositories (MSR 2004), International Conference on Software Engineering*. IET, 2004, pp. 90–94.

[16] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proceedings International Conference on Software Maintenance (ICSM 2003)*. Los Alamitos CA: IEEE Computer Society Press, Sep. 2003, pp. 23–32.

[17] ——, "Analyzing and relating bug report data for feature tracking," in *Proceedings IEEE Working Conference on Reverse Engineering (WCRE 2003)*. Los Alamitos CA: IEEE Computer Society Press, Nov. 2003, pp. 90–99.

[18] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 433–436. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321702

[19] A. Begel and R. DeLine, "Codebook: Social networking over code," in *ICSE Companion*, 2009, pp. 263–266.

[20] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: discovering and exploiting relationships in software repositories," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 125–134.

[21] A. Begel and T. Zimmermann, "Keeping up with your friends: Function foo, library bar.dll, and work item 24," in *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, ser. Web2SE '10. New York, NY, USA: ACM, 2010, pp. 20–23. [Online]. Available: http://doi.acm.org/10.1145/1809198.1809205

[22] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 966–969.

[23] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 62.

[24] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[25] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: a taxonomical study of large commits," in *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 2008, pp. 99–108.

[26] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 97–106.

[27] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V.Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fixing datasets," in *Proceedings of the 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering and the 12nd European Software Engineering Conference*. ESEC-FSE, 2009, pp. 121-130.

[28] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and the 13rd European Software Engineering Conference*. ESEC-FSE, 2011, pp. 1525.

[29] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR, 2013, pp 121-130.

[30] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. ASE, 2016, pp 8190.

[31] C. Williams and J. Spacco, "SZZ revisited: verifying when changes induce fixes," in *Proceedings of the 2008 workshop on Defects in large software systems*. Proceeding DEFECTS 2008, pp 32-36.