2016 IEEE International Conference on Software Quality, Reliability and Security

Which Change Sets in Git Repositories Are Related?

Jasmin Ramadani
University of Stuttgart
jasmin.ramadani@informatik.uni-stuttgart.de

Stefan Wagner
University of Stuttgart
stefan.wagner@informatik.uni-stuttgart.de
ORCID: 0000-0002-5256-8429

***Abstract*—Software repositories contain valuable information about the history of software changes. Using data mining, researchers have identified file changes that happened together frequently to present hints for necessary changes to developers. However, not all file change sets are related. This can affect the recommendations about coupled file changes negatively by delivering irrelevant couplings to the developers. The commit time and branching characteristics of Git have not been investigated together in previous heuristics for grouping related change sets. We exploit the mappings between commit messages and issue ids for judging the relatedness of change sets. We propose a heuristic for Git and investigate the influence of two factors, the time between the commits and their branching on the relatedness of change sets using the repositories of five open-source systems using logistic regression. According to our findings, the combination of these two factors influences the relatedness of change sets. Individually measured, only the time significantly influences the relatedness, the branching itself does not. Our results support previous heuristic that also in Git repositories the commit time is important for grouping related change sets.**

I. Introduction

Version control systems store information concerning which files were changed, when they were changed, who made the change and what the files contained before the change [18]. Two main types of version control systems are used today: centralized version control systems (CVCS) like CVS or Subversion and distributed version control systems (DVCS) like Git or Mercurial. In CVCS we have a central server holding the version database where the developers check out their projects on their local computers. In DVCS each team member has the complete repository on local machine called local repository. It contains the entire project and its history. The developers can clone the remote repository into their local repositories and commit in the local repository.

In many version control systems, especially in DVCS, file changes are organized in commits or change sets. In the commits, the information about the commit time and the branching is preserved. Branching is used to separate commit changes from each other.

Software change issues like features or bugs can be tracked using issue tracking systems [8]. Every issue contains specific information about the problems we need to solve like id, description, type, status and other attributes.

Over time, the amount of data in the repositories becomes very large. To learn from it, we need a technique to extract useful information stored in it. Frequently, data mining is used to analyze software repositories. Mining software repositories (MSR) is a term that has been coined to describe the investigation of software repositories using data mining [19], [28].

Coupled changes describe a situation where the developer changes a file and also changes another file shortly afterwards. Using MSR we can find the files in the repositories that changed frequently together in the past and propose change couplings to the developers for bug fixing or maintenance tasks.

A. Problem Statement

Coupled file changes can be found in different change sets. However, not all file change sets are related. Files can be changed on several occasions dealing with different issues. We identify the relatedness of file change sets, meaning that we report only those including changes having the same change context.

Several heuristics for grouping file changes have been defined in the literature [16], [20]. They are used to group related change sets in repositories which support atomic change sets. They use factors like the developer who committed the file changes and the time of commit. However, there is no heuristic to examine repositories in DVCS like Git which is a very popular versioning system. The committing and the branching as an important DVCS feature, have not been previously investigated together to find related file change sets.

Although there are also CVCS that support committing and branching, the fact the developers in Git work locally and commit their changes on the remote server, motivates us to investigate how these factors influence the relatedness of change sets.

Not examining these DVCS characteristics can lead to the grouping of unrelated change sets suggesting irrelevant coupled file changes to the developers.

B. Research Objective

The overall goal is to improve the coupled change suggestions generated from DVCS data. The aim of this research is to examine a heuristic for Git to group related file change sets in a repository for extracting coupled file changes. We include important DVCS characteristics like the dealing with the time of commit and the commit location in the branches.

978-1-5090-4127-5/16 $31.00 © 2016 IEEE
DOI 10.1109/QRS.2016.52

C. Contribution

We present a quasi-experiment where we extract related change sets from Git repositories. The basic idea is that change sets are related if they are associated with the same issue. We use a mapping of commit messages with the issue ids to identify related change sets. We measure the time between the commits and their branching status to define a heuristic to find related change sets. We investigate the influence of these two factors on their relatedness using logistic regression and can show that their combination has a significant influence on the relatedness of change sets, whereby individually measured, only the commit time has an significant influence on the relatedness.

D. Context

We use Git repositories from five different open source software project repositories. Two of the projects are open-source projects from the University of Stuttgart. The rest of the projects are open source projects on GitHub[1].

II. Background

A. Git Version Control

Git[2] is one of the most used DVCS today. A Git repository provides a copy of all files in the repository and a copy of the repository we work with [21]. It organizes the changed files in commits which represent atomic change sets.

The commits are identified by commit id hashes, the attributes describing the committer, commit date and commit message. Git does not store a snapshot of all files. If one file changes, the complete repository changes [23]. Developers can fetch changes from the remote repository, pull to merge the changes in the current repository branch and push changes to the remote repository [23].

An important attribute of Git is the way it handles with the time of changes in the source code. Since Git does not track the timestamps when the files were originally modified, it does not have an central time concept, it tracks only the time of commit. Branches exist also in other versioning systems and are very often used in Git. This allows the developers to create many lines of development. To be able to leverage the branches, we investigate the Git graph, a visual presentation of branches in the repository.

B. Coupled Changes

One of the most popular data mining techniques is the discovery of frequent item sets. Identifying item sets performed together frequently is one of the most basic tasks in data mining [15]. Developers change various source files with different frequency during software development. As transactions, we define the commits consisting of different files. Let us have the following three transactions: $T_1 = \{f_1, f_2, f_3, f_7\}$, $T_2 = \{f_1, f_3, f_5, f_6\}$, $T_3 = \{f_1, f_2, f_3, f_8\}$. From these three transactions, we identify the rule that files f_1 and f_3 are frequently changed together: f_1 and f_3 are coupled. This means that when the developers changed file f_1, they also changed file f_3. If these files are changed together frequently, it can help other persons by giving them a recommendation that if they change f_1, they should also change f_3. Let $F = \{f_1, f_2, ..., f_d\}$ be the set of all items (files) f in a transaction and $T = \{t_1, t_2, ..., t_n\}$ be the set of all transactions t. Each transaction contains a subset of chosen items from F called item set. The support count δ is an important property of an item set, reporting the number of transactions containing an item. If the item-sets have a support threshold greater than a minimum specified by the user, we say they are frequent.

C. File Change Sets

Older versioning systems like CVS, do not maintain change sets. The information which software artifacts were checked in together is lost. Therefore, researchers investigate the change history using the technique of fixed or sliding time windows [10], [12], [29]. In a versioning system that provides atomic change sets, such as Subversion or Git, commits represent the atomic change sets. They include the basic set of files used to extract coupled file changes.

The number of commits containing particular coupled file changes identifies how frequent these changes are. The fact that these couplings are spread through the repository and committed by different authors in different time periods introduces the need for grouping these change sets to find the related ones. For this purpose, we need a proper heuristic.

III. Related Work

There is a lot of scientific work dedicated to investigating software repositories to find logically coupled changes, e.g. [3], [9], [11]. If we analyze the used methodology, most of the studies investigating coupled changes use some kind of data mining for this purpose [12], [17], [20], [25], [26], [28], [29]. Here, the frequent item-sets technique is used to identify frequent changes in the source code [20], [28], [29]. This data mining technique uses various algorithms to determine the frequency of these changes. Sometimes the Apriori algorithm is used [20], [29], however other algorithms like the FP-Tree algorithm are also in use [28].

If we consider the type of version control systems investigated, the majority of the studies use CVCS typically CVS or Subversion as a data source for their analysis. DVCS, especially Git, are very popular in practice and have specific characteristics how the data is tracked, related to the grouping of changed files and tracking the time of change. To our knowledge, there are few studies investigating Git repositories [4], [6], [13].

Different heuristics have been proposed in the literature for related change sets. Heuristics based on the data source and pruning technique are introduced in [16] where several techniques are introduced to group and reduce the change sets. Entity data, developer data, name similarity data are considered and pruned by their frequency, recency or randomization.

[1] https://github.com/
[2] http://git-scm.com/

Kagdi et al. in [20] suggest *time interval, committer* and *+time interval+committer* heuristics. The time interval heuristic includes file changes committed by different committers in a predefined period of time. The total number of time periods identifies the number of change sets groups.

Both studies propose heuristics and then evaluate them classically using precision and recall measures.

For our analysis, we use the *committer* heuristic to generate the starting set of coupled file changes. We add afterwards two more factors: the time between the commits and the branching location. For our new heuristic we consider the developer, the time of commit and the branching of the commits. We identify this heuristic as *committer+time+branch*.

Very few studies deal with investigating characteristics of commit messages. They concentrate on creating vocabulary terms [1] and the words that appear in the messages [7].

The studies we presented in this section miss a complete solution for reporting coupled changes considering Git characteristics. Unlike other studies we do not simply propose a heuristic, we investigate the influence on the relatedness using the following two factors: time between the commits and the branching. We examine their influence on the grouping of related change sets.

IV. Related Change Sets

A heuristic is necessary to identify groups of related change sets from which we can extract coupled file changes. The same file changes can be found in different commits performed on different occasions by various developers. In a set of coupled file changes, although we deal with the same files, the commit messages can describe different changes which are not necessarily related (Table I). Here the commit messages describe totally different issues. This means that also unrelated changes happen to be grouped together as coupled file change suggestions.

TABLE I
UNRELATED CHANGE SETS MESSAGES

Commit 1	Commit 2
synchronizing layouts done refs #827	added highlighter to the connection anchors refs #347

TABLE II
RELATED CHANGE SETS MESSAGES

Commit 1	Commit 2
began to adapt controller structure refs #503	Adapt controller structure refs #503
added icons to export wizard refs #868	added new icons and new png filter to export wizard refs #868

To avoid this, we need to identify the related change sets out of which we can extract the coupled file changes.

Previous studies test the relatedness of file changes by checking whether they are part of the solution for some task. The performance is usually based on the calculated precision and recall [16], [20].

So what makes change sets related? The developers want relevant suggestions. We assume this is the case if all of the file changes deal with a single task. The change set can have various granularities and composition across different tasks [20]. One task can be solved in several steps in more than one commit. In case a task cannot be solved in a short time, the work on it can be interrupted and continued at another time. Here, a single issue id can be found in several commits. The first example in Table II shows that the task is addressed in two steps, the developer started dealing with the controller structure in one occasion and finished the changes later in another one.

The same change can be committed many times in different occasions. The second example in Table II, the change set commented as "added icons to export wizard" and the change set commented as "added new icons and new png filter to export wizard" represent the same file changes repeatedly in different commits. We assume that the change sets are related if they are addressing the same issue either solving the issue in many steps or repeating the change in several occasions.

The comments stored as commit messages in Git contain some description about the changed files committed in the version system and describe the change purpose [22]. However, the commit messages do not always deliver understandable textual content. The analysis of these messages even with the help of natural language processing is not always useful. The use of merge points to map the commit messages and the issues from the issue tracking system is a useful practice today. Here, the commit messages contain the issue ids which identify a particular task, a feature or a bug. We use these merge points to compare the issue ids in the commit messages and investigate the relatedness of change sets.

There are three cases of mapping commits and issues: one commit mapped to one issue, where the commit and issue are not used in further mappings, one commit mapped to several issues, a quite rare case where one change can lead to another change in a similar issue and many commits mapped to one issue which represents the most considered mapping.

V. Experimental Design

In this section we define the research questions, hypotheses and metrics used in our analysis.

We select our metrics using the GQM approach [2] and its MEDEA extension [5]. Our **goal** is to define a heuristic for related change sets in Git. Our **objective** is to determine the relation of the commit time and branching towards the relatedness of change sets. The **purpose** is to measure the relatedness for different time commit and branching values. Our **viewpoint** is as software developers and the targeted **environment** is open source systems.

A. Research Questions

We investigate the influence of the time between the commits and branching location of commits on the relatedness of change sets. For that purpose we define the following research questions:

RQ1: Is there an influence of the time between the commits and the existence of branching on the relatedness of file change sets? This question is relevant to investigate since Git maintains the time of commit of file changes and supports local and remote branching in the development. This is our main research question. We investigate the combination of these two factors on the relatedness of change sets which leads us to the formulation of the heuristic we proposed.

Considering the fact that we have two factors which can influence the relatedness, we define two additional research questions questions:

RQ2: Is there an influence of the time between the commits on the relatedness of file change sets? Here we refer to the first individual factor, the commit time. We investigate different time periods between the commits to find out if this influences their relatedness.

RQ3: Is there an influence of the existence on branching on the relatedness of file change sets? We concentrate on the second factor, the branching location. We investigate how the commit location in the same or different branches takes effect on their relatedness.

Additionally we are interested to see if there is a difference in the influence of these two factors across the projects we investigate.

RQ4: Is there any difference in the relatedness of the time between the commits and the branching on the relatedness of change sets across projects. We investigate the spread out of the relatedness for every repository individually to explore how it varies across the projects.

B. Hypotheses

We formulate the following hypotheses to answer the research questions in our study.

For **RQ1** we define the following hypotheses:

$\mathbf{H_{0.1}}$: There is no influence of the time between the commits and branching on the relatedness of file change sets.

$\mathbf{H_{A.1}}$: There is an influence of the time between the commits and branching on the relatedness of file change sets.

To answer **RQ2** we formulate these hypotheses:

$\mathbf{H_{0.2}}$: There is no influence of the time between the commits on the relatedness of file change sets.

$\mathbf{H_{A.2}}$: There is an influence of the time between the commits on the relatedness of file change sets.

For **RQ3** we derive the hypotheses as follows:

$\mathbf{H_{0.3}}$: There is no influence of the branching on the relatedness of file changes-sets

$\mathbf{H_{A.3}}$: There is an influence of the branching on the relatedness of file change sets.

C. Experimental Variables

1) Independent Variables: In this experiment, we define two independent variables: *time between commits* and *branching*. The first independent variable, represents a continuous numerical variable, measuring the time between a pair of commits. We use calendar days as measure for this variable. The second independent variable is a dichotomous variable having two categorical states representing the branching status: are the commits in the same branch or not.

2) Dependent Variables: We have one dependent variable called *relatedness of file changes*. This variable is also dichotomous and has two categorical values: related and not related.

D. Experiment Design

The specific type of the variables directly influences the type of our experiment design. We use an experiment to investigate the effects of two independent variables (continuous and categorical) on a single dependent categorical variable. When the independent variables are either continuous and/or categorical, we have a *regression* layout [24]. If the dependent variable is categorical we use *logistic regression* [14].

This is an extended situation as we have multiple independent variables measured along with a single dependent variable so we use *multiple logistic regression*.

E. Objects

The prerequisite to include the repositories for analysis is that most of their commits must contain references to the issue ids. We use Git repositories of five different open-source software projects. The first project, ASTPA[3], is an Eclipse RCP application. The second, RIOT[4] is a Java web services and Android project. Project number three is Metrics[5], a JavaScript library for visualizing time-series data. The fourth project is Akka[6], a toolkit for message-driven applications on the JVM. Project number five is an Add-on builder for Firefox[7]. The first two projects were found in the local GitLab on the University of Stuttgart. The other projects are popular projects hosted on GitHub.

F. Experiment Instruments

We collect the log data from Git repositories using a self-developed program written in Java. We use it to automatically extract the commits, their attributes, like commit messages and commit times, as well as the committed file. This data is stored in a relational database. We use the SPMF[8] data mining framework to generate the coupled file changes. We have adjusted the framework to work with databases instead of text files. The data analysis is performed using the statistical software SPSS[9].

G. Data Collection Procedure

First of all we extract the logs from the repository. We gather the commits, the files changed and the attributes for all the developers who committed the file changes and store them in the database.

We prepare the log data for data mining whereby we exclude the empty commits, the commits containing single entries and

[3]http://sourceforge.net/projects/astpa/
[4]https://github.com/SE-Stuttgart/riot/
[5]https://github.com/mozilla/metrics-graphics
[6]https://github.com/akka/akka
[7]https://github.com/mozilla/addon-sdk
[8]http://www.philippe-fournier-viger.com/spmf/
[9]http://www-01.ibm.com/software/analytics/spss/products/statistics/

the commits which do not contain references to the issue ids in the commit messages. We do not consider the data from those developers who did less than 50 commits. This filtering is performed to set a rule for the minimum frequency of coupled file changes. In this case, we set the minimum frequency to be 5. We set a minimum support level of 10% meaning that we do not consider the file changes found in less than 5 commits. This way we have a user-set degree of the frequency of coupled changes.

For every developer we choose single random set of coupled file changes. We join the appropriate set of attributes to the coupled file changes. Using our scripts for automated Git log extraction, we enlist the information about the committer, commit time, commit message and files changed. We take the chosen coupled file change and pair all the commits where this change was detected. The pairs are generated by combining all the change sets for a specific coupled change. We start with the latest commits and continue with previous commits. For example, if a set of files changed together was found in the commits with ids of *6c08c5a, e7c56dd, cfc90b3* and *9d29bd5*, we will examine the relatedness of all the combined pairs of commits.

H. Analysis Procedure

1) Commit Time Analysis: Every commit in Git has its own time stamp marking the time of commit in the repository. We calculate the difference of time between the paired commits for the investigated set of coupled file changes. The time difference is stored in a data sheet for every commit pair entry included in the analysis. We use calendar days as time units for our measurement.

2) Branching Analysis: We analyze the placement of the commits considering their branching. We leverage the branch of the investigated commit and compare its position with the second commit in the pair. If they are found on the same branch, we set the value of the branching variable to yes, otherwise if they are committed in different branches, we denote it as no.

3) Relatedness Analysis: To find related change sets, we analyze the messages content for all possible pairs of commits where the file change coupling was found.

We parse the commit message text for mappings with issue ids. For the first commit in the pair, we look up the issue id in the commit message text.

Next, we repeat this for the second commit in the pair. To determine if these two commits are related, we compare their ids. If in both commit messages, the references to the issues match, we denote them as related. Commit pairs with different issue references are classified as not related.

4) Logistic Regression Analysis: We follow the procedure for statistical analysis with logistic regression presented by Schwab in [27] which includes the determining the sample size, possible numerical problems, the relationship between the combination of the independent variables and the dependent variable, the relationship between the individual independent variables and the dependent variables, the strength of logistic regression relationship and the logistic regression model validation. The description of all steps is available at http://dx.doi.org/10.5281/zenodo.49187.

The ratio of the dependent variable values is 1 to 4 in favor of the negative values to the positive ones for the relatedness. This ratio produces over-fitting of the regression model. To avoid this, we perform down-sampling where we divide the population of the negative values for the dependent variable in double size subsets than the number of positive values subsets for the dependent variable. We use 80-20 cross strategy to validate the regression model.

VI. Results and Discussion

The data sets for the study results are available at http://dx.doi.org/10.5281/zenodo.49187.

A. Descriptive Statistics

The summary of the descriptive statistics for the experiment are presented in Table III.

From 1641 commit pairs from all five projects, after the removal of commit messages without issue references, there are 1218 left for analysis. We have 136 related and 1082 unrelated pairs of commits.

We also report the minimum, maximum, mean, mode and standard deviation for the time between the related commits. The minimum commit time difference between the related change sets is 0 or in a single day. The maximum varies over the five projects between 4 and 25 days, whereby these values are very rare and extreme. Calculating the mean, we found that the average time difference between two related commits varies between 0.35 and 3.33 days. The overall value is around 2 days. The value of the standard deviation varies between 0.933 and 6.457 days depending on the project. The standard deviation value for all the related change sets is around 3.17. This value is close to the mean. Hence, we have a low spread of the commit dates for the related change sets. The mode shows that the most frequent difference in the time of change for the related change sets is between 0 and 1 days. For the complete data set it is 0 days, which means that most of the related change sets were committed together during one day.

The relatedness distribution is presented in Table IV. For the time between the commits, we have created two groups based on the mean value which is 2 days. The first group includes the change sets where the commit time difference is less or equal than 2 days. The second group includes the change set where the commit time difference is more than two days.

From 136 pairs of related change sets, 97 commits or 71.3% were committed in less than 2 days, 39 commits or 28.6% of them were committed in more than 2 days. Here we see that most of the related change-sets were committed in less than two days difference.

The relatedness distribution of the unrelated change sets, 64 or 5.9% of the unrelated commits happened in less than 2 days, 1018 or 94.1% were committed in more than 2 days. Almost 95 percent of the unrelated change sets were committed in more than two days difference.

TABLE III
DESCRIPTIVE STATISTICS

Project	Commit pairs	Commit pairs related	Commit pairs unrelated	Time between related commits in days				
				min	max	mean	st. dev	mode
ASTPA	520	36	484	0	15	1.94	2.714	1
RIOT	149	48	191	0	25	3.19	4.088	1
Metrics	70	20	50	0	11	2.1	3.478	1
Akka	201	20	81	0	4	0.35	0.933	0
Mozilla Addon	365	12	266	0	22	3.33	6.457	0
All Projects	1218	136	1082	0	25	2.13	3.17	0

TABLE IV
RELATEDNESS DISTRIBUTIONS

	Related	Not related
Time between commits $\leq$ 2 days	97 (71.3%)	64 (5.9%)
Time between commits $>$ 2 days	39 (28.6%)	1018 (94.1%)
Commits in the same branch	55 (40.4%)	49 (4.6%)
Commits not in the same branch	81 (59.6%)	1033 (95.4%)

Considering the branching of related change sets, 55 commits or 40% are in the same branch, 80 or near 60% are not in the same branch. In both cases, in the same and in different branches, we have related change sets. For the unrelated changes, 49 of them or 4.6% are in the same branch, 1033 commits or 95,4% are not. Most of the unrelated change sets were found in different branches.

TABLE V
REGRESSION RESULTS

Statistic	Average	Average (validated)
Sample Size	270	227
Model Chi-Square	0.000	0.000
Standard error Time	0.043	0.048
Standard error Branch	0.441	0.488
B Coeff. (Time)	-0.300	-0.291
B Coeff. (Branch)	-0.607	-0.566
p Wald (Time)	0.000	0.000
p Wald (Branch)	0.194	0.221
Exp (B) (Time)	0.741	0.742
Exp (B) (Branch)	0.553	0.761
By chance Accuracy	66.5%	66%
Model Accuracy	90.4%	91.6%

B. Influence of the time between the commits and the branching on the relatedness

To answer RQ_1, we test our main hypothesis investigating if the combination of time and the branching influences the relatedness of the coupled changes. The regression results are presented in Table V.

The average sample size for all data sets after the down-sampling is 270. This value is much larger than the minimum number of 10 cases per variable, which satisfies the requirements for the sample size. The average value of the standard error for both of the independent variables is lower than the 2.0 threshold which reports that there are no numerical problems in the analysis.

The presence of a relationship between the combination of the independent variables and the dependent variable is based on the statistical significance of the model-chi square. Our analysis shows that the model chi-square value is 0.000 which is far less than the 0.05 threshold. Therefore the null hypothesis is rejected, meaning that the combination of time between the commits and the branching has an influence on the relatedness of the change sets. The values of the model chi-square statistical significance presented in Table V for the validated subsets are very close to the values for the full data set. This satisfies the classification accuracy of our regression model.

The values in Table IV indicates that related change sets were found in both groups for the time between the commits. Also they were found both in the same or in different branches. This means that a combination of these two factors influences the relatedness. According to the results, the possibility to have related change sets drops with the rise of the time of commit and the placement of the commits in different branches.

C. Influence of time between the commits on the relatedness

For RQ_2, we test our second hypothesis where we examine the influence of the time between commits on the relatedness of change sets. The average value for the Wald test for the time variable as independent variable is 0.000 which is lower than the 0.05 threshold. this result reports a significant presence of a relationship between the individual independent variable (time between the commits) and the dependent variable (relatedness), rejecting the null hypothesis in this case.

The average B coefficient value is negative, meaning that one unit change in the time has a negative influence on the relatedness odds. The average value of the exponent of the B coefficient Exp (B) for the time of commit is 0.741 (0.742 for the validated data set) which means that a change of one unit in the commit time when the other independent variable is constant is going to decrease the odds to have related change sets by 26%.

Indeed the results in table IV show that the average commit time period between related change sets is two days, whereby most frequently, related change sets were committed during one day. The frequency of the commit time differences between related change sets is presented in Figure 1. Here we can see that the number of related change sets cases drops with the increase of the time. This confirms the high influence of

the time between the commits on the relatedness of change sets.

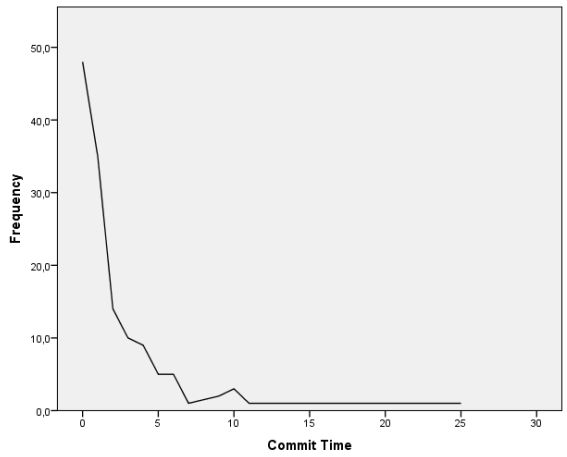

Fig. 1. related change sets time distribution

D. Influence of the branching on the relatedness

For RQ_3, we test the null hypothesis about the influence on the branching on the relatedness of change sets. The average value of the Wald test statistics for the data set is 0.194 and for the validated subsets is 0.221. These values are larger than 0.05. Therefore, the null hypothesis in this case is not rejected. This means that individually measured, the branching does not influence the relatedness of change sets when the other variable is constant.

The results show that both in the same or in different branches we have related and unrelated changes as presented in table IV. There is no clear distribution between the related and unrelated change sets considering the branch location of the commits.

E. Influence of the time between the commits and branching on the relatedness across projects

Regarding RQ_4, we explore if there is a difference in the influence of these two factors on the relatedness across the five project repositories. We use the mean and the standard deviation values to investigate the spread of the Wald Statistic values for all five projects. The results presented in Table VI show that the data from the first three projects delivers useful data. The last two projects do not deliver data ready for analysis. The Wald statistic value which is equal to 0 for all three projects, reports that there is a strong relation between the time of commit and the relatedness. The value of the Wald statistics for the branching in all three projects is above the 0.05 threshold which identifies that there is no significant influence of the branching on the relatedness of the change sets.

The mean value of the Wald Statistics for the commit is 0.000 for all projects. This shows that across all projects, the time has a significant influence on the relatedness of the change sets. For all projects, the value of the Wald Test for the branching is larger than 0.05. This means that in all five projects, the branching does not influence the relatedness of the change sets. The standard deviation for both factors identifies low spread meaning that there is no a significant difference in the influence across the projects.

The last two projects do not deliver measurable interaction between the time of commit and the branching because that there is no variance in the branching. In these projects, all unrelated cases of change sets are found in different branches. There is not a single case of an unrelated change set where the commits are in the same branch. This make the regression analysis not possible. A possible reason for this situation could be that these two projects include heavy branching. This may influence the way the related change sets are organized in branches. To be able to further investigate this kind of projects, we need to analyze a larger number of change sets per project.

TABLE VI
INFLUENCE ON RELATEDNESS ACROSS PROJECT

Project	Wald Test (Time)	Wald Test (Branch)
ASTPA	0.000	0.396
RIOT	0.000	0.157
Metrics	0.000	0.310
Akka	-	-
Firefox Addon	-	-
Mean	0	0.287
St. Deviation	0	0.157

VII. THREATS TO VALIDITY

The mapping between the commits and the issues represents a central construct validity threat for our study whereby the developers could provide false references. Using data from different projects and various developers having high rate of commit mappings decreases the possibility for this threat.

The relatively high data mining support threshold of 10% excludes a number of coupled changes and commits. However, this threshold ensures a relatively high level of frequency of the reported coupled file changes which avoids the possibility to have changes that could happen by chance.

The influence of the experimenter during the execution of the experiment could affect the internal validity of the study. The experimenter needs to define the relatedness of the change sets by manually examining the commit messages. We use an additional review of a sample of the relatedness of the file changes by a second person.

A potential internal threat could also be the influence of particular developer data on the change sets for the analysis. The commit behavior of the developers and the discipline in the referencing of the commits with the issue can influence the truthfulness of this relation. We minimize the influence on the results by randomly selecting a single set of commits per developer before the logistic regression analysis has been performed.

An external validity threat is the limitation of our analysis generalization on other projects. The analysis is performed on projects which include mapping between the commits and the issues. However, on GitHub there are many projects where this mapping is implemented. We have also used different project repositories developed in various environments. Although we have used relatively small repositories, the use of well known analysis methods ensures that we can repeat the analysis on larger projects.

VIII. Conclusion and Future Work

We have investigated five Git repositories to determine the influence of time between the commits and the branching on the relatedness of change sets. The regression results give evidence that the combination of these two factors influences the relatedness. This supports the heuristic we proposed. Individually measured, the time of commit has a significant influence on the relatedness. Our findings show that most of the related change sets are committed in single day. The branching itself did not play a significant role towards the relatedness. The values for these factors measured across the projects show that in cases of heavily branched projects, we need more coupled change cases for analysis. Our results confirm the heuristics presented in [19] where the developer and the time of commit groups related change sets. We support the influence on the time factor on the relatedness of change sets.

Our study targets Git which is not particularly investigated for mining repositories. We give evidence that time does influence the relatedness of change sets, yet the branching does not have this relation in our results. The *developer+time* heuristic can be implemented in a tool to recommend coupled file changes for the developers during maintenance of bug-fix tasks. Using this heuristics we can decrease the number of change sets we investigate for coupled changes and lower the time effort for the data mining analysis.

Our approach limitation is the content of the commit messages we analyzed. We do not involve comments which do not include the issue ids. Also the size of the repositories and the number of committers limits the number of coupled changes and the change sets we investigate.

The next steps in our research is to investigate larger repositories and examine higher number of change sets for a deeper investigation about the relatedness of change sets.

References

[1] A. Alali, H. Kagdi, and J. I. Maletic. What's a typical commit? a characterization of open source software repositories. *International Conference on Program Comprehension*, 0:182–191, 2008.

[2] V. R. Basili, G. Caldiera, and H. D. Rombach. *The Goal Question Metric Approach*. Wiley, 1994.

[3] J. M. Bieman, A. A. Andrews, and H. J. Yang. Understanding change-proneness in oo software through visualization. In *IWPC*, pages 44–53, 2003.

[4] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, and P. T. Devanbu. The promises and perils of mining git. In *MSR*, pages 1–10, 2009.

[5] L. Briand, S. Morasca, and V. Basili. An operational process for goal-driven definition of measures. *IEEE Transactions on Software Engineering*, 28:1106–1125, 2002.

[6] E. Carlsson. Mining git repositories : An introduction to repository mining, 2013.

[7] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *ICSE*, pages 422–431. IEEE / ACM, 2013.

[8] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM*, pages 23–, 2003.

[9] B. Fluri, H. Gall, and M. Pinzger. Fine-grained analysis of change couplings. In *SCAM*, pages 66–74, 2005.

[10] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *ICSM*, pages 190–197, 1998.

[11] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *IWPSE*, pages 13–23, 2003.

[12] D. German. Mining CVS repositories, the softChange experience. In *MSR*, pages 17–21, 2004.

[13] D. German, B. Adams, and A. Hassan. Continuously mining distributed version control systems: an empirical study of how linux uses git. *Empirical Software Engineering*, pages 1–40, 2015.

[14] N. J. Gotelli and A. M. Ellison. *A Primer of Ecological Statistics*. Sinauer Associates, 2004.

[15] J. Han, M. Kamber, and J. Pei. *Data mining concepts and techniques, third edition*. Morgan Kaufmann Publishers, 2012.

[16] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In *ICSM*, pages 284–293, 2004.

[17] L. Hattori, G. P. dos Santos Jr., F. Cardoso, and M. C. Sampaio. Mining software repositories for software change impact analysis: A case study. In *SBBD*, pages 210–223, 2008.

[18] K. Hinsen, K. Laeufer, and G. K. Thiruvathukal. Essential tools: Version control systems. *Computing in Science and Engineering*, 11(6):84–91, 2009.

[19] H. H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance*, 19(2):77–131, 2007.

[20] H. H. Kagdi, S. Yusuf, and J. I. Maletic. Mining sequences of changed-files from version histories. In *MSR*, pages 47–53, 2006.

[21] J. Loeliger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, Inc., 1st edition, 2009.

[22] A. Mauczka, M. Huber, C. Schanes, W. Schramm, M. Bernhart, and T. Grechenig. Tracing your maintenance work - a cross-project validation of an automated classification dictionary for commit messages. In J. de Lara and A. Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 301–315. Springer, 2012.

[23] S. Otte. Version control systems.

[24] G. P. Quinn and M. J. Keough. *Experimental design and data analysis for biologists*. Cambridge University Press, Cambridge, UK, New York, 2002. 10e réimpr. en 2010.

[25] F. V. Rysselberghe and S. Demeyer. Mining version control systems for facs. In *MSR*, pages 48–52, 2004.

[26] J. Sayyad-Shirabad, T. Lethbridge, and S. Matwin. Mining the maintenance history of a legacy software system. In *ICSM*, pages 95–104, 2003.

[27] J. Schwab. Logistic regression - complete problems. University Lecture, 2003.

[28] A. T. T. Ying, G. C. Murphy, R. T. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[29] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.