The Journal of Systems and Software 128 (2017) 220–235

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Using contextual information to predict co-changes

Igor Scaliante Wiese[a,*], Reginaldo Ré[a], Igor Steinmacher[a], Rodrigo Takashi Kuroda[b], Gustavo Ansaldi Oliva[c], Christoph Treude[c], Marco Aurélio Gerosa[c]

[a] Computer Science Department, Federal University of Technology – Parana (UTFPR), Brazil
[b] PPGI/PPGI – UTFPR/CP, Federal University of Technology – Parana (UTFPR), Brazil
[c] Department of Computer Science, University of São Paulo (USP), São Paulo, SP, Brazil

ARTICLE INFO

Article history:
Received 17 September 2015
Revised 7 July 2016
Accepted 10 July 2016
Available online 12 July 2016

Keywords:
Contextual information
Co-change prediction
Software change context
Change coupling
Change propagation
Change impact analysis

ABSTRACT

Background: Co-change prediction makes developers aware of which artifacts will change together with the artifact they are working on. In the past, researchers relied on structural analysis to build prediction models. More recently, hybrid approaches relying on historical information and textual analysis have been proposed. Despite the advances in the area, software developers still do not use these approaches widely, presumably because of the number of false recommendations. We conjecture that the contextual information of software changes collected from issues, developers' communication, and commit metadata captures the change patterns of software artifacts and can improve the prediction models. **Objective**: Our goal is to develop more accurate co-change prediction models by using contextual information from software changes. **Method**: We selected pairs of files based on relevant association rules and built a prediction model for each pair relying on their associated contextual information. We evaluated our approach on two open source projects, namely Apache CXF and Derby. Besides calculating model accuracy metrics, we also performed a feature selection analysis to identify the best predictors when characterizing co-changes and to reduce overfitting. **Results**: Our models presented low rates of false negatives (~8% average rate) and false positives (~11% average rate). We obtained prediction models with AUC values ranging from 0.89 to 1.00 and our models outperformed association rules, our baseline model, when we compared their precision values. Commit-related metrics were the most frequently selected ones for both projects. On average, 6 out of 23 metrics were necessary to build the classifiers. **Conclusions**: Prediction models based on contextual information from software changes are accurate and, consequently, they can be used to support software maintenance and evolution, warning developers when they miss relevant artifacts while performing a software change.

1. Introduction

Software systems are built upon artifacts that depend on one another. As a result, some artifacts evolve together throughout software development (Canfora et al., 2014). Co-change prediction has been used to support developers while they work on change requests (Bohner and Arnold, 1996; Zimmermann et al., 2005). For example, if a developer modifies a file without knowing that this file should be changed along with another file (e.g., a configuration file), the change will be incomplete, which might end up introducing defects (Hassan and Holt, 2004). Predicting co-changes can be useful to avoid incomplete changes, notifying developers about artifacts that are likely to change together.

Previous research has proposed different approaches to predict co-changes based on dynamic (Orso et al., 2004) and static analysis (Briand et al., 1999), frequent past changes and change coupling analysis (Canfora et al., 2010; Gall et al., 1998; Ying et al., 2004; Zimmermann et al., 2005), and conceptual analysis (Gethers and Poshyvanyk, 2010; Revelle et al., 2011). Other approaches combined these different techniques into hybrid methods (Dit et al., 2014; Gethers et al., 2012; Hassan and Holt, 2004; Kagdi et al., 2013; Malik and Hassan, 2008). Although significant progress has been made, these approaches still suffer from low accuracy. In practice, this means these approaches would both issue several false recommendations and miss many relevant items.

Given this challenge, a possible solution to improve the accuracy of co-change prediction is to use information sources un-

* Corresponding author.
E-mail addresses: igor.wiese@gmail.com, igor@utfpr.edu.br (I.S. Wiese), reginaldo@utfpr.edu.br (R. Ré), igorfs@utfpr.edu.br (I. Steinmacher), rodrigokuroda@gmail.com (R.T. Kuroda), goliva@ime.usp.br (G.A. Oliva), ctreude@ime.usp.br (C. Treude), gerosa@ime.usp.br (M.A. Gerosa).

http://dx.doi.org/10.1016/j.jss.2016.07.016
0164-1212/

explored so far. We conjecture that using contextual information from software changes in prediction models – something not done by current approaches – can reduce the number of false recommendations and improve the effectiveness (e.g., accuracy) of co-change prediction.

Our motivation to use contextual information is related to the fact that software artifacts are changed for different reasons (Oliva and Gerosa, 2015a; Oliva et al., 2013) and the context involved in these changes can better describe the conditions in which two artifacts are more prone to co-change. For this purpose, we built prediction models *for each* pair of artifacts using contextual information from one release to predict if the two artifacts would change together in issues of the consecutive release.

To illustrate our approach, consider the following concrete example from our analysis. For the release 2.0 of the CXF project, the files *JMSConduit.java* and *JMSOldConfigHolder.java* changed together in 19 commits (in 17 different issues). In another 26 commits, the former did not change together with the latter. Collecting contextual information for each commit made to *JMSConduit.java* for release 2.0, we were able to predict, for release 2.1, 15 co-changes between both files and 18 commits in which the former did not change together with the latter. In only two commits the predictions were wrong. For this pair, the most important contextual information was the number of source code lines changed in *JMSConduit.java*, the number of words used to describe the issues, and who reported each issue.

In summary, our goal in this paper was to investigate novel sources of information to improve co-change prediction, focusing on traces of contextual information produced during software evolution. Therefore, the central elements of our approach comprise human- and process-centric contextual information sources (e.g., issue comments, developers' communication network, commit practices, and historical information about artifact changes), rather than language/artifact-centric information sources (e.g., static and dynamic dependencies, such as call graphs).

Our hypothesis is that co-changes can be better predicted using socio-technical context instead of only using different types of coupling among artifacts, since developers need to interpret the reported problem, choose artifacts to modify, and collaborate with other developers (De Souza et al., 2007; Schröter et al., 2012; Tsay et al., 2014; Wolf et al., 2009b; Zanetti, 2012). We set out to investigate the following central question: *Can contextual information extracted from issues, developers' communication, and commit metadata improve the accuracy of co-change prediction?*

By analyzing two open source projects, Apache Derby (11 releases) and Apache CXF (8 releases), we addressed the following two research questions:

(RQ1) Can we accurately predict co-changes between two artifacts using contextual information from issues, developers' communication, and commit metadata?

Previous work has shown that prediction models can be built to predict co-change occurrence (Zimmermann et al., 2005). However, we conjecture that such models can be improved by using contextual information from issues, developers' communication, and commit metadata. Models that bring fewer false positives and false negatives are necessary to avoid misleading alerts to developers.

(RQ2) What are the most important kinds of contextual information when predicting co-changes?

Knowing which kind of contextual information is the most important indicator of co-change can help practitioners during software evolution and maintenance tasks. From a practical perspective, selecting the best subset of metrics can be useful to reduce the effort to create a prediction model, since less data needs to be collected and fewer metrics need to be computed.

We relied on the concept of *change coupling* to select pairs of files to evaluate (Oliva and Gerosa, 2015b). We compared our prediction model to an association rule model. The association rule model is widely used in the literature and can be considered a baseline model (Ball et al., 1997; Ceccarelli et al., 2010; Dit et al., 2014; Ying et al., 2004; Zimmermann et al., 2005).

Overall, we found that contextual information extracted from issues, developers' communication, and commit metadata enables a highly accurate prediction of co-changes. We observed that the average rates of false positives and false negatives were lower than 11% and 8% respectively. These results suggest that our model can be leveraged for the development of novel co-change prediction tools to support software evolution and maintenance.

The remainder of the paper is organized as follows. Section 2 presents a discussion about contextual information from software changes. Section 3 describes our case study design, while Section 4 presents the results with respect to our two research questions. Section 5 presents related work. Section 6 discusses the results and their implications on practice and research. Finally, Section 7 presents the threats to the validity of our study, and Section 8 presents conclusions, limitations, and our plans for future work.

2. Contextual information from software changes

According to Dey et al. (2001), context is "*any information that can be used to characterize the situation of entities*." To Brézillon and colleagues (2004), context represents "*a complex description of shared knowledge about physical, social, historical, or other circumstances within which an action or an event occurs*."

In line with these definitions of context, we conjecture that software changes can be characterized by the circumstances in which they occur. In our study, these circumstances encompass the meta-data of the change request, communication that occurs in the issue tracker system, and the commit meta-data. In other words, these three sources of information represent our notion of context for software changes (and co-changes).

Many authors have indicated that software changes are influenced by socio-technical aspects (Bettenburg and Hassan, 2012; De Santana et al., 2013; Kwan et al., 2012; Leano et al., 2014; Schröter et al., 2012; Tsay et al., 2014; Zanetti, 2012). In fact, we consider these socio-technical aspects to be contextual information of software changes, since developers' communication and collaboration revolve around software change requests (issues). Hence, in this paper we investigate three sources of contextual information, namely: issues, developers' communication, and commits. It is important to highlight that we are not claiming that contextual information can be the cause of co-changes, as we are just exploring the prediction power of these kinds of information.

Table 1 presents the definition and the rationale for picking each contextual metric used in this work. We also grouped each metric into a contextual dimension. In the results section, we discuss also the performance of each contextual dimension, since some metrics sources might not be available in some projects. This set of metric was inspired by a systematic mapping we previously conducted (Wiese et al., 2014a) and by systematic reviews on prediction models applied in software engineering (Hall et al., 2012; Radjenović et al., 2013).

In the next Section, we present the descriptive statistics of the contextual information and we also provide details about how the contextual information was used to build the prediction models.

3. Case study design

In this section, we present the studied software projects, and the data extraction and analysis approaches. In Fig. 1 we present an overview of the data extraction and prediction/analysis approaches. In the data extraction phase we present the way we

Table 1
Contextual information dimensions and their metrics.

Contextual dimension	Metric name	Type	Description	Rationale
Issue Context (IC)	Issue type	String	Type of an issue, chosen when the issue was opened. The values can be: Bug, Improvement, New Task, Documentation, Infrastructure, etc.	Changes resolve issues of different types. Some co-changes are more likely to happen when fixing a bug, while others can appear when implementing new features (Zimmermann et al., 2012).
	Was issue reopened?	Boolean	If the status of an issue was ever "reopened", then we set this value to "TRUE"	An issue may be reopened especially when the change request is complex and the number of files involved to fix the issue is high, thus some specific co-changes may appear (Shihab et al., 2010).
	Issue assignee	String	The name of the developer assigned to resolve an issue	The assignee can work on issues related to specific parts of the software. In this sense, assignees can be more prone to submit patches that change the same set of files (Zhang et al., 2013).
	Issue reporter	String	The name of the stakeholder that reported the issue	An issue reported by the same reporter might involve the same files, since the reporter might be interested in some specific requirements (Zimmermann et al., 2012).
Communication Context (CC)	# of Issue comments	Numeric	Number of comments posted to an issue	The communication dimension involves aspects from discussion around issues. Patterns of this discussion can indicate when files change together. For example, some co-changes can happen in issues with more discussion, more messages, and more words (wordiness) because the issue is difficult to understand, or the files necessary to fix this issue are more complex to understand (Bettenburg and Hassan, 2012).
	# of Issue discussants	Numeric	Number of distinct stakeholders that commented on an issue	
	Issue wordiness	Numeric	Amount of words in the discussion. We only considered words with more than 2 characters.	
	# of Issue developer commenters	Numeric	The number of distinct developers that committed the file in a previous release and comment on an issue in which the file changed in the current release	
Developer's Role in Communication (DRC)**	Betweenness centrality *	Numeric	Number of times a node acts as a bridge along the shortest path. We used the median of betweenness among all developers that commented on an issue in which the pair of files changed	Centrality measures can capture developers' roles in the communication. We hypothesize that developers involved in a discussion have different values of Betweenness and Closeness, indicating that the roles can be used to predict co-changes. Previous work has shown the importance of these metrics in other software engineering problems (Bicer et al., 2011; Bird et al., 2009; Wolf et al., 2009a).
	Closeness centrality *	Numeric	Closeness can be regarded as a measure of how long something will take to spread in a network. We used the median of closeness among all developers that comment on an issue in which the pair of files changed	
Structural Hole of Communication (SH)**	Constraint *	Numeric	Measures the lack of holes among neighbors. This measure is based on the degree of exclusive connections. Low values indicate that there are few alternatives to access a single neighbor	Structural hole metrics denote gaps between nodes in a social network and represent that people on either side of the hole have access to different flows of information, indicating that there is a diversity of information flow in the network. In previous work (Wiese et al., 2014c), we successfully used structural holes to identify recurrent change couplings. In this sense, these metrics represent a way to analyze the communication network revolving around the software co-changes.
	Hierarchy *	Numeric	Measures the constraint to a single node. High values indicate that the constraint is concentrated in a single node neighbor	
	Effective Size *	Numeric	Measures the portion of non-redundant neighbors of a node. High values represent that many nodes among the node's neighbors are not redundant	
	Efficiency *	Numeric	Efficiency normalizes the effective size by the number of node neighbors. High values show that many neighbors are non-redundant	
Communication Network Properties (NP)**	Size *	Numeric	The number of nodes in the network	Based on Conway's Law (Conway, 1968; Kwan et al., 2012), which describes the relation between communication and software architecture, communication network properties may predict co-changes.
	Ties *	Numeric	Number of edges in the network	
	Diameter *	Numeric	The diameter is defined to be the maximum, over all pairs of vertices (u,v), of the length of the shortest path from u to v	
	Density *	Numeric	Density is calculated as the percentage of the existing connections to all possible connections in the network	
Commit Context (ComC)	Committer	String	The name of the developer that committed the file	Developers can contribute to specific parts of the software project and they can always commit the same files. This indicator can be useful to identify when two files are committed together by the same set of committers (Bird et al., 2011; Mockus, 2010).
	Is same ownership?	Boolean	The ownership of a file is computed in a previous release. We ranked the developers per number of commits for each file. The Top 10% of developers in this ranked list were considered "owners." We set the value to "TRUE" if the actual committer is part of the ranked ownership list	
	# of lines of code added	Numeric	Sum of lines of code added in a commit	Code churn or a specific operation (add or delete) on lines of codes can indicate specific aspects for different co-changes (D'Ambros et al., 2012; Moser et al., 2008; Rahman and Devanbu, 2013).
	# of lines of code deleted	Numeric	Sum of lines of code deleted in a commit	
	Code Churn	Numeric	Sum of number of lines added and deleted	

* Details about how to implement the SNA metrics can be found in the related papers cited (Bettenburg and Hassan, 2012; Bird et al., 2009; Meneely et al., 2008; Wolf et al., 2009a), or in (Wasserman and Faust, 1994). To compute the SNA metrics we used the JUNG Java framework:http://jung.sourceforge.net/.

** To compute the communication network, we used each developer as node and each message as arc. More details can be found in our previous work (Wiese et al., 2014b,2014c)

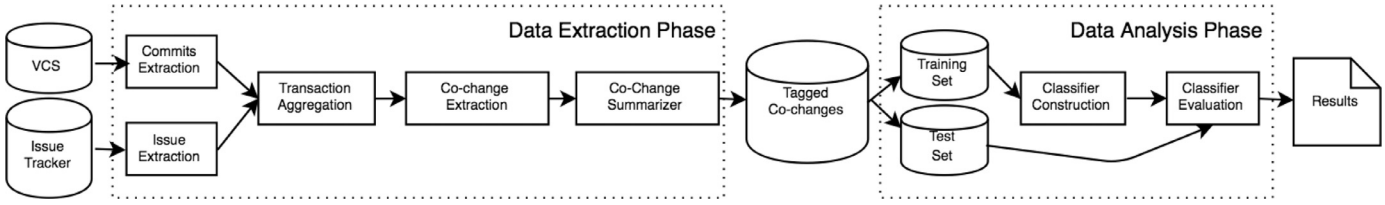

Fig. 1. An overview of our data extraction and prediction/analysis approaches.

Table 2
Characteristics of the studied projects.

Project		Apache CXF	Apache Derby
General information	Domain	Web services framework	Database
	Period of analysis	2007-08-30 to 2015-02-19	2004-10-04 to 2015-01-21
	Release cycle (average time in days) (Min-Max)	992 ± 232 (stdev) (699-1555)	2501 ± 884 (stdev) (469–3266)
	Release versions studied	2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, and 3.0	10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9, 10.10, 10.11 and 10.12
Data extraction phase summarization	Number of fixed issues associated with a studied version	3095 of 6175 issues collected	3459 of 6787 issues collected
	Number of commits associated with an issue	8975 of 20,288 commits collected	8386 of 10,701 commits collected
	Average of fixed issues per version (Min–Max)	657 ± 201(stdev) (215–950)	404 ± 222 (stdev) (31–808)
	Average of commits per version (Min–Max)	1860 ± 815 (stdev) (344–3025)	1172 ± 603 (stdev) (54–2213)
	Ratio of issues analyzed per total of fixed issues	19.06% (590 of 3095)	20.21% (699 of 3459)
	Ratio of commits analyzed per total of commits	19.35% (1737 of 8975)	20.24% (1697 of 8386)
Relevant pairs of files summarization	Number of distinct pairs of files analyzed by project	58 relevant Pairs of files	90 relevant Pairs of files
	Total of co-changes Java-Java, Java-Test, Java-XML	155, 17, 17	188, 44, 35
	Total of co-changes Test-Test, Test-XML	1, 0	25, 7
	Total of co-changes Xml-Xml	10	1
	Avg., min. and max. of support per version	2% of Issues ± Stdev (1%) (1–4%)	3% of Issues ± Stdev (1%) (2%-7%)
	Avg., min. and max. of confidence per version	78% ± Stdev (20%) (71%–85%)	74% ± Stdev (22%) (58%–99%)
	Number of defects associated with studied co-changes	22.37% (132 of 590 Issues)	22.32% (156 of 699 Issues)

collected and preprocessed the data, and how we found relevant co-changes. In the prediction/analysis phase, we describe how we used the contextual information to build the prediction models and how we evaluated these models.

3.1. Studied software projects

In order to address our research questions, we studied two Apache projects: CXF and Derby. Apache CXF is an open source web services framework. CXF helps developers to build and develop services using frontend programming APIs, like JAX-WS and JAX-RS. Apache Derby is an open source relational database implemented entirely in Java based on the JDBC protocol and the SQL language.

Table 2 provides an overview of the studied systems. The table is split in three parts. The first part describes the project characteristics in terms of domain, period of analysis, the average number of days of each release cycle, and the releases that we applied our approach to. The second and third parts show results from the data extraction phase (more details in the next section). Both projects were implemented in the Java language. Even though both projects are Apache projects, it is important to mention that they have different developers who follow different practices to change the code. Currently, Derby[1] has decreased the amount of commits submitted. Conversely, CXF[2] has increased the amount of commits and new contributors to the project according to OpenHub.net.

Table 3 presents the descriptive statistics for each numeric metric in all releases analyzed for each project. We observe differences between the two projects with regards to each contextual dimension presented in Table 1. An analysis of median values for the metrics from the CC, DRC, SH and NP dimensions showed that Apache CXF presented fewer social interactions. For example, the median of *issue wordiness* in Derby was 8 times higher than CXF's median.

In Apache Derby, developers and users participated more in the discussions (higher values of *# of issue discussants* and *# of issue developer commenters*). Because of this, the SNA metrics computed in the DRC, SH and NP dimensions were also higher in Derby than in CXF. Considering software changes, the median of lines of code removed and code churn in CXF were higher than in Derby. However, the median of lines of code added in Derby was higher than in CXF. This shows two distinct evolution patterns: while Derby had more code added during its evolution, CXF's changes often involved removing and modifying lines of code.

Hence, the studied systems are of different sizes and domains. Some of their metric values were substantially different, revealing distinct evolution patterns.

3.2. Data extraction phase

We used two data sources to conduct this study: Issue Tracking Systems (ITS) and Version Control Systems (VCS). An *issue* is a development task, such as a bug report or a new feature request. Several commits may be necessary to close an issue. We collected the logs containing all the commits made by developers to an issue, and aggregated them into change transactions.

Issues are often logged in an Issue Tracking System, like Bugzilla or JIRA, and have a unique identifier (ID). This ID helps to identify the commits associated with an issue. We extracted data related to the issues of the studied software projects in order to address our research questions. As can be observed in Fig. 1, the data extraction phase (DE) was split into 4 steps. In the following, we briefly describe each of these steps.

[1] https://www.openhub.net/p/cxf.
[2] https://www.openhub.net/p/derby.

Table 3
The descriptive statistics for the numeric metrics listed in Table 1..

Dimension	Metric	Apache CXF						Apache Derby					
		Min.	Max.	Avg.	Std.	Sum	Median	Min.	Max.	Avg.	Std.	Sum	Median
CC	# of issue comments	0	47	2.72	4.74	2238	1	0	175	15.16	17.53	19,680	10
	# of issue discussants	0	6	1.11	1.13	513	1	0	13	3.38	1.90	4386	3
	Issue wordiness	0	10,705	365.31	801.94	300,654	122	8	24,132	1539.47	2318.47	1998,231	806
	# of issue developer commenters	0	4	0.62	0.69	513	1	0	7	1.64	1.10	2129	2
DRC	Betweenness centrality	0	1.95	0.03	0.16	21.28	0	0	7.3	0.37	0.76	482.2	0
	Closeness centrality	0	1	0.15	0.26	126.51	0	0	1	0.26	0.24	335	0.2
SH	Constraint	0	1.26	0.21	0.4	175.65	0	0	1.62	0.61	0.42	792.57	0.76
	Hierarchy	0	1	0.32	0.45	264.39	0	0	1	0.54	0.33	696.34	0.51
	Effective size	0	4.32	0.65	0.93	534	0	0	7.67	2.21	1.37	2865.44	2
	Efficiency	0	1	0.34	0.45	280.5	0	0	1	0.6	0.31	782.61	0.64
NP	Size	0	6	1.11	1.13	911	1	0	13	3.38	1.9	4386	3
	Ties	0	19	1.05	2.24	866	0	0	103	8.58	11.77	11,143	5
	Diameter	0	3	0.42	0.57	342	0	0	5	1.27	0.73	1642	1
	Density	0.5	1	0.91	0.18	751.85	1	0.5	1	0.82	0.18	1059.74	0.83
ComC	# of lines of code added	0	1	0.64	0.29	6672.28	0.67	0	1	0.72	0.3	8304.28	0.81
	# of lines of code deleted	0	1	0.3	0.25	3106.72	0.29	0	1	0.28	0.3	3165.72	0.18
	Code churn	0	2836	50.41	121.75	522,982	20	0	13,944	53.61	260.91	616,871	13

Proj – project, Rel – release versions, CC – communication context, DRC – developer role communication, NP – network properties , ComC – commit context.

(**DE Step 1) Commit and issue extraction.** We used the CVSAnalY[3] tool to collect and parse commits from the VCS. We extracted all commits, including commit author, timestamps, and commit message metadata. We used Bicho[4] to parse and collect all issues from JIRA.

(DE Step 2) Transaction aggregation. An issue can be resolved after several commits. To avoid missing cases of co-changes, we grouped commits that addressed the same issue. To link commits to issues, we looked for identifiers of issues (issue number) in the commit message. Apache projects usually adopt the following pattern [ProjectName-IssueNumber] to indicate that a specific commit needed to be applied to fix an issue.

For example, consider the issue 4850 from the JIRA repository of Apache CXF.[5] We could link it to 4 commits using the pattern in the commit message. One of the commit messages was: "*[CXF-4850] add WebMethod (operationName=) to the inherit method to avoid name conflict in wsdl*". For Derby, a very similar pattern is used. The only difference is that developers do not use the brackets to highlight the project and issue number. An example is the message of the commit[6] submitted to fix issue LUCENE-4850[7] in Derby: "*LUCENE-4850: Upgrade Clover version to 3.1.10 (to allow Java 7)*".

We also checked whether the commits were made while the issue was carrying the status *open* and moved to the status *fixed* afterwards. Commits made to issues in any other status (e.g invalid, duplicated, won't fix, not a problem and unresolved) were dismissed since this source code was not integrated in the project. We found that 470 commits were made in 244 issues with a status other than open in Derby, and 272 commits in 106 issues in CXF. We can conclude that the number of commits and issues discarded cannot affect the training and test sets significantly, since we found less than 6% of the commits collected in issues with a status other than open in the whole history of both projects.

(DE Step 3) Finding strongly coupled pairs of files. To select the pairs of files, we mined the change history of each release of each project. We considered just the changes (commits) submitted as patches to issues. In cases where an issue had more than one commit associated with it, we grouped all the commits into one single change transaction by taking the union of the files in the change-sets. After this preprocessing stage, we selected the pairs of files by picking the top 25 most relevant association rules. In the following, we describe what association rules are and how we calculated their relevance.

An association rule is an implication of the form $I \Rightarrow J$, where I and J are two disjoint sets of items (a.k.a., item sets). A relevant rule $I \Rightarrow J$ means that when I occurs, J is likely to co-occur. In other words, it implies that transactions that contain I are likely to contain J as well. In the scope of this study, a rule $I \Rightarrow J$ means that J is change-coupled to I. We also consider that I and J are file sets composed by one single file, where $I = \{f_i\}$ and $J = \{f_j\}$ and $f_i \neq f_j$. The relevance of association rules can be measured according to several metrics. In this study, we employed the metrics of *support* and *confidence*, which have been extensively used in previous Software Engineering research studies (Canfora et al., 2010; Kagdi et al., 2013; Zimmermann et al., 2005).

The support of a rule $r = I \Rightarrow J$ corresponds to the number of transactions that contain both I and J. Therefore, support determines how *evident* a rule is. In turn, confidence is often interpreted as the *strength* of a rule. More specifically, given the same rule r, its confidence refers to the fraction of transactions containing I where J also appears. Relevant rules thus have high values for support and confidence. In the scope of this study, support and confidence were calculated as follows:

$$\begin{aligned}
&\textbf{\textit{support}}(\mathbf{r}) = support(I \Rightarrow J)\\
&\quad = support(I \cup J) = support\left(\{f_i\} \cup \{f_j\}\right)\\
&\quad = \text{the number of transactions that contain both } f_i \text{ and } f_j\\
&\textbf{\textit{confidence}}(\mathbf{r}) = confidence(I \Rightarrow J)\\
&\quad = \frac{support(I \Rightarrow J)}{support(I)} = \frac{support(I \cup J)}{support(I)} = \frac{support\left(\{f_i\} \cup \{f_j\}\right)}{support(\{f_i\})}\\
&\quad = \text{fraction of transactions containing } f_i \text{ where } f_j \text{ also appears}
\end{aligned}$$

For each release, we calculated all possible rules involving pairs of files. Afterward, for each pair of rules $<r_1 = I \Rightarrow J, r_2 = J \Rightarrow I>$, we discarded the one with lower confidence (their support is always the same). Next, we sorted all remaining rules according to support and used confidence as a tiebreaker. We then selected the

[3] https://github.com/MetricsGrimoire/CVSAnalY.
[4] https://github.com/MetricsGrimoire/Bicho.
[5] https://issues.apache.org/jira/browse/CXF-4850.
[6] http://svn.apache.org/viewvc?view=revision&revision=1457809.
[7] https://issues.apache.org/jira/browse/LUCENE-4850.

Table 4
An example of fragment of training/test set.

Pair of files	Commit ID	# Issue	Set of metrics from contextual information (issues, developers' communication and commit)	Class
JMSConduit.java – JMSOldConfigHolder.java	1	1760	Issue Type = Bug, Issue Reopened = 0, Assignee = ffang, Reporter = ffang, # of commenters = 3, # of dev commenters = 2, wordiness = 438…	0
JMSConduit.java – JMSOldConfigHolder.java	13	1773	Issue Type = Improvement, Issue Reopened = 0, Assignee = njiang, Reporter = chris@die-schneider.net, # of commenters = 4, # of dev commenters = 3,, wordiness = 576…	0
JMSConduit.java – JMSOldConfigHolder.java	1450	1832	Issue Type = Improvement, Issue Reopened = 0, Assignee = chris@die-schneider.net, Reporter = chris@die-schneider.net, # of commenters = 3, # of dev commenters = 2, wordiness = 764…	1
JMSConduit.java – JMSOldConfigHolder.java	1700	2207	Issue Type = Bug, Issue Reopened = 0, Assignee = njiang, Reporter = liucong, # of commenters = 2, # of dev commenters = 1, wordiness = 311…	0
JMSConduit.java – JMSOldConfigHolder.java	1701	2207	Issue Type = Bug, Issue Reopened = 0, Assignee = njiang, Reporter = liucong, # of commenters = 2, # of dev commenters = 1, wordiness = 311…	0
JMSConduit.java – JMSOldConfigHolder.java	2115	2316	Issue Type = Improvement, Issue Reopened = 0, Assignee = dkulp, Reporter = marat, # of commenters = 1, # of dev commenters = 0, wordiness = 43…	1
JMSConduit.java – JMSOldConfigHolder.java	…	…	…	…

top 25 rules, which we deemed as the pairs of files to be analyzed in that release.

In Table 2 we show details about the pairs of files selected by association rules. For example, in the CXF project, the rules comprise 58 unique pairs of files from 9 releases. In Derby, 90 pair of files were selected from 12 releases. Applying the prediction models to these co-changes we covered almost 22% of the defects for each project and almost 20% of the commits made in the projects. Our main reason to study the strongly coupled pairs of files is that we want to compare the models based on contextual information with association rules. According to Zimmermann et al. (2005) association rules can be useful to recommend co-changes when they have high values of confidence and support values of 2% for the transactions analyzed. A similar justification can be also found in Bavota et al. (2013).

In Table 2 we also reported that the pairs of files used in CXF occurred in an average of 2% of the issues, and 3% in Derby considering the support value, and, we obtained rules with at least, 70% of confidence on average. For this reason, we decided to pick only the top 25 most relevant association rules in each release analyzed.

(DE Step 4) Co-change summarizer. We recovered all issues and commits from each pair of files selected in the previous step. This process was followed for each release version. In this step, we computed the metrics for contextual information of issue reports, developers' communication, and commit metadata for each co-change found in our list of the 25 most relevant association rules.

Table 4 presents an example of the metrics computed for each co-change. We collected the metrics to build the training and test sets when the issues were carrying the status *closed/fixed*. The column "Pair of files" indicates the pair of files analyzed. In this example, file I (JMSConduit.java) is the Left File, and file J (JMSOldConfigHolder.java) is the Right File identified by the relevant association rule. As we mentioned, all metrics were computed to file I. In this sense, the set of metrics was extracted from each commit and issue (indicated in columns "Commit" and "Issue") when the file I (JMSConduit.java) was changed.

Tagging co-changes. In order to assess the co-change prediction, we built training and test sets to compare the results of our approach with the actual changes that occurred in each project. To tag each co-change, we looked at the Right File.

Each commit containing file I (Left File) was checked, and when the commit propagated changes to file J (Right File), we assigned the commit to class "1″. Otherwise, we assigned it to class "0″ to indicate that only file I was changed in a certain issue. Hence, the value 1 indicates that the Left File (file I) and the Right File (file J) were committed together. The lines of Table 4 indicate that file I had 6 commits in 5 distinct issues. Rows 3 and 6 show the examples of change propagation between files I and J (Class column = 1).

3.3. Data analysis phase (prediction task)

On the right-hand side of Fig. 1, we present an overview of the Data Analysis Phase. In this phase, we performed and evaluated the prediction. Classifiers were built using the training set, and their performance was evaluated on issues in the test set. We briefly describe each step in our analysis below.

(DA Step 1) Classifier construction. To build the classifiers we need to use a training set. For each release of Apache CXF and Derby, we generate a .csv file similar to Table 4 to use as training or test set. The release version N is used as training set and the release version N+1 is used as test set. We run the *random forest technique* to construct classifiers to predict the co-changes.

Considering the advantages, Random forests are frequently used in classification problems since the models are fast, scalable, and it is not necessary to worry about tuning a number of parameters like in other machine learning algorithms. The models can be used with large and small datasets, and also can handle problems of missing data (Breiman, 2001).

The random forest technique was already used in many different areas like biology/medicine, pattern recognition, traffic control, image classification (Breiman, 2001), and in software engineering (McIntosh et al., 2014). Since we did not know the size of the projects that we could apply our approach to, how much training data all pairs of files identified in those projects, we chose the random forest algorithm to build our models.

The random forest technique builds a large number of decision trees at training time using a random subset of all the attributes. In our study, these attributes correspond to the contextual information from issue report, issue communication, and commit metadata.

The technique performs a random split to ensure that all of the trees have a low correlation between them (Breiman, 2001). Using an aggregation of votes from all trees, the random forest technique decides whether the final score is higher than a chosen threshold to determine if a co-change will be predicted as true or not.

We construct classifiers using all metrics described in Table 1. In order to check which predictor has the best performance, we run a feature selection algorithm combined with random forest classifiers. We implement our classifiers using the R package *Caret* (Kuhn, 2008).

Feature selection analysis: We performed the feature selection analysis using the Recursive Feature Selection (RFE), also using the Caret R package (Kuhn, 2008). The RFE function implements a backward selection of predictors based on predictor importance

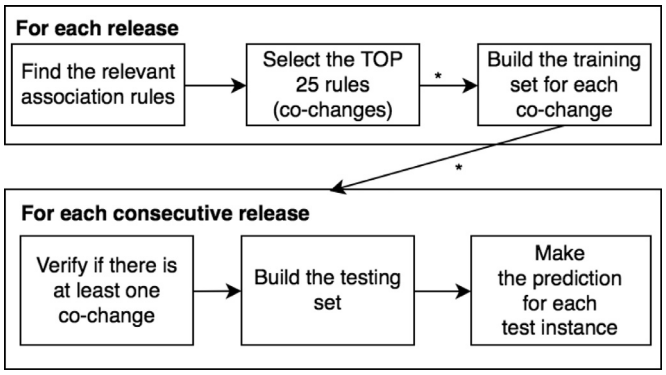

Fig. 2. Classifier evaluation steps.

Table 5
The confusion matrix.

Predict as	Observed as	
	changed	Not changed
Changed	TP (true positive)	FP (false positive)
Not changed	FN (false negative)	TN (false positive)

ranking. The predictors were ranked and the less important ones were sequentially eliminated prior to modeling.

The RFE algorithm can be used for predictor selection, model fitting, and performance evaluation. We used the training data with the RFE algorithm to select the predictors. According to Kuhn (2008) the RFE algorithm is useful to improve the model interpretability, decreasing the training times and enhancing generalization by reducing overfitting.

(DA Step 2) Classifier evaluation. To evaluate our classifier, we used training and test sets generated for each release of each project studied. The steps are summarized in Fig. 2.

Basically, for each release, we found the relevant association rules and we selected the top 25 relevant rules. Each one of these 25 rules is related to a pair of files that was analyzed. Each pair of files has at least one co-change in the analyzed release. Using data from one release we built a training set. In the consecutive releases, we verified if there was at least one co-change between the pair of files and then we built the test set to make the predictions.

A concrete example. The pair of files JMSConduit.java and JMSOldConfigHolder.java was found in our list of 25 most relevant association rules in release 2.0. We found at least a single co-change between these two files in the consecutive release (2.1), thus, we built a test set for release 2.1 as a .csv file – similar to Table 4 – and used it to make the predictions and evaluate our model.

After the evaluation, we moved to the next release (2.2) to find if there is at least one co-change between JMSConduit.java and JMSOldConfigHolder.java. If it is true, we used the data from release 2.1 as training set and the data from release 2.2 as test set. If it is false, we discarded the release as test set. For example, in this case, as in the release 2.3 we did not find these files being changed together, we stopped to create training and test sets for JMSConduit.java and JMSOldConfigHolder.java.

The performance of the classifier was measured in terms of recall, precision, area under the curve (AUC) and Mathews correlation coefficient (MCC). To compute these metrics it is necessary to use a confusion matrix, as shown in Table 5.

Recall (R): We calculated recall to identify the proportion of instances that the model nominated for changing together and that actually changed. To obtain the recall value, we used the following formula: TP/TP+FN.

Precision (P): We measured precision to identify the rate of co-changes that have actually changed together. To obtain the precision value, we used the following formula: TP/TP+FP.

F-measure (F1): is the harmonic mean of precision and recall. We used the following formula: 2*(precision*recall)/(precision + recall).

Area Under the Curve (AUC): The area under the curve that plots true positive rate against the false positive rate, for various values of the chosen threshold used to determine whether a file is classified as part of a co-change or not. Values of AUC range from 0 (worst performance) to 1 (best performance).

Mathews Correlation Coefficient (MCC): MCC is a correlation coefficient between the observed and predicted classification (Powers, 2011). This measure takes into account TP, FP, TN, and FN values and it is generally regarded as a balanced measure, which can be used even if the classes are unbalanced.

Matthews Correlation Coefficient (MCC) is used to show the quality of our predictions. For binary classification tasks, MCC has attracted the attention of the machine learning community as a method to summarize the confusion matrix into a single value (Baldi et al., 2000). An MCC coefficient of + 1 represents a perfect prediction and 0 means a random prediction. We calculated MCC using the following expression (Powers, 2011):

$$\text{MCC} = \frac{(TP * TN) - (FP * FN)}{\sqrt{(TP+FP) * (TP+FN) * (TN+FP) * (TN+FN)}}$$

For the results discussion, we report the AUC, MCC, precision, recall and the confusion matrix values.

4. Results

The main goal of this work was to evaluate the potential of contextual information gathered from issues and commit metadata to predict co-changes between two files. In the following sections, we discuss the results for each research question.

Using data from 9 releases from Apache CXF we covered 19.06% of the fixed issues from their software history, considering 19.35% of the commits. For Apache Derby we collected data from 20.21% of the fixed issues and 20.24% of the commits. It is important to highlight that these values are low because we are filtering and testing only the most frequent co-changes in each release.

Considering the co-changes identified by association rules (our top 25 ranked pairs) used as input to make the predictions, we found that 77.5% of the pairs of files inspected in Apache CXF are co-changes between Java files. For Apache Derby, we detected that 62.7% of the pairs of files also involved co-changes between Java files. Few co-changes occurred between configuration files (e.g. xml files) and test files. Considering the number of defects, the co-changes were associated with 22% of the defects from each project. The percentage of defects is expressive if we consider that the defects are related to 58 distinct co-changes that were identified in Apache CXF and 90 distinct relevant pairs of files in Apache Derby.

4.1. (RQ1) Can we accurately predict co-changes between two artifacts using contextual information from issues, developers' communication, and commit metadata?

Approach. We built one classifier for each co-change identified for each release of Apache CXF and Derby. We measured the performance of each classifier analyzing the precision, recall, F1, AUC and MCC. We also used the confusion matrix to evaluate the false positive and false negative rates, since high values of these two measures can reduce the usability of the classifiers in practice.

Results. Our classifiers improve the precision values compared to the association rules model and vastly outperform random classifiers for both projects.

Table 6
Confusion matrix by release for all co-changes.

Project	Release version	Confusion matrix			
		TP	FP	FN	TN
Apache CXF	2.1	237	29	21	292
	2.2	125	14	13	243
	2.3	23	6	2	28
	2.4	195	19	21	543
	2.5	150	13	24	382
	2.6	96	10	28	217
	2.7	135	18	23	262
	3.0	86	10	5	134
	Total	**1047**	**119**	**137**	**2101**
Apache Derby	10.2	264	6	19	721
	10.3	62	9	15	195
	10.4	327	37	25	875
	10.5	83	12	16	214
	10.6	166	13	31	626
	10.7	92	5	16	250
	10.8	54	2	3	97
	10.9	262	18	31	662
	10.10	101	10	16	219
	10.11	63	3	11	86
	10.12	15	0	1	16
	Total	**1489**	**115**	**184**	**3961**

In Table 6, we present a more fine-grained analysis using the confusion matrix for each test set in each project version. We report values of TP, FP, FN and TN.

Results in absolute terms showed that, for Apache CXF, we predicted 1047 correct couplings. Our models presented 119 (10.25%) false positives, predicted as changed when the files did not change together. For Apache Derby, we obtained 1489 correct commits against 115 false positives (7.16%).

As our models were designed to predict when a co-change will occur, we also can present the results considering when the co-change didn't happen and just one file changed. In these cases, for Apache CXF we obtained 2101 correct predictions for the Left file when the Right file did not change. We obtained 137 false negatives (6.12%). For Apache Derby, we obtained 3961 correct predictions against 184 false negatives (4.43%).

These results are particularly interesting because false positive values can cause overhead to developers if the approach was used in a real scenario. For example, with many false positives, a classifier could predict that Left file and Right file changed together when they would actually not change, warning a developer at least to take a look at Right file without need, causing an unnecessary overhead in a developer task.

Another important result was related to the small amount of false negatives reported. In these few cases, classifiers could predict just a change for Left file, forgetting to alert the developer to change the Right file, causing possible software defects or future maintenance.

4.1.1. Does the contextual information improve the precision and recall values compared to a baseline model?

Previous work (Canfora et al., 2010) reported precision values between 70% and 90%, while the recall values were reported often between 25% and 10% when co-change prediction is done using association rules models. On the other hand, a direct comparison is difficult to do because the techniques were used to predict co-changes of different sets of artifacts in different projects.

However, to make the comparison fairer, since we are predicting co-changes considering two files, we re-implemented the association rule algorithm used in the literature (Canfora et al., 2010; Kagdi et al., 2013; Zimmermann et al., 2005) and used it as baseline model to compare our results. We used the association rules as described in Section 3 - (DE Step 3) Finding strongly coupled pairs of files.

Table 7
Comparsion of recall and precision values.

Project	Our approach			Association rules		
	R	P	F1	R	P	F1
CXF	88.42%	89.79%	89.09%	100%**	34.25%	51.02%
Derby	89.00%	92.83%	90.87%	100%**	30.22%	46.41%

Since each co-change has a rule indicating that this pair of files was prone to change in the consecutive release, we used this recommendation and assume (with support and confidence range listed in Table 2), that all issues involving the pairs within the co-change set were prone to change in the consecutive release. Because of this, Table 7 presents the values of recall of our Baseline Model as 100% (**). On the other hand, the precision values of association rules are lower, because all changes related to file I did not propagate to file J, thus, the association rules obtained many false positives recommending changes when they did not occur.

In Table 7 we note that the average precision was improved. The precision values (P) obtained by our models were 89.79% and 92.83% for Apache CXF and Derby, respectively. Considering the Baseline Model recall values, we note that our classifiers obtained a difference of 55.54% for Apache CXF and 62.61% for Apache Derby in terms of precision.

Even considering a "perfect" predictor of co-changes as our baseline model, the difference between the recall values (R) was lower than the improvements in precision values. Considering the F-measure (F1) values we also note differences in both projects. For CXF we obtained 89.09% against 51.02% for the baseline model, and for Derby we got 90.87% vs 46.41%.

Qualitative inspection. To give more details about the comparison of contextual information we inspect the results and describe the practical aspects related to each approach describing a real case from the CXF project.

Calculating the frequency of past changes (association rules), we find that jaxrs/client/ClientProxyImpl.java and jaxrs/interceptor/JAXRSOutInterceptor.java changed together in 14 commits during release 2.6. Analyzing ClientProxyImpl.java's changes, we find that it did not changed with JAXRSOutInterceptor.java in 35 commits. We could infer that this coupling is strong after analyzing the distribution of co-changes and finding that this pair of files was found in our TOP 25 list.

Based on this coupling information it seems reasonable to infer that both files are prone to change together in the consecutive release (2.7). For release 2.7, coupling-based approaches would correctly predict co-changes in 12 commits. However, in another 16 commits, ClientProxyImpl.java changed in the release 2.7, but without a corresponding change in JAXRSOutInterceptor. Analyzing all commits involving ClientProxyImpl.java in version 2.7, approaches based on coupling would always recommend a co-change with JAXRSOutInterceptor.java, since this is the only association rule found in the software history related to ClientProxyImpl.java. In this case, the coupling-based approaches would raise 16 false positives, in which commits affected ClientProxyImpl.java but not JAXRSOutInterceptor.java.

In our approach, we collect contextual information for each commit to ClientProxyImpl.java for release 2.6. Based on this information we are able to predict, for release 2.7, 8 co-changes between both files and 16 commits in which ClientProxyImpl.java changed, but not JAXRSOutInterceptor.java. Our approach gives 4 false positive recommendations. For this pair, the most influential contextual information was the number of lines added, the number of lines deleted and amount of words in the discussion.

Table 8
similarity over releases comparing the number of co-changes among the pairs of files analyzed.

Project	Release version	Similarity (Jaccard index)
Apache CXF	2.0/2.1	0.70
	2.1/2.2	0.38
	2.2/2.3	0.40
	2.3/2.4	0.61
	2.4/2.5	0.44
	2.5/2.6	0.65
	2.6/2.7	0.73
	2.7/3.0	0.75
	Median	**0.63**
Apache Derby	10.1/10.2	0.60
	10.2/10.3	0.60
	10.3/10.4	0.40
	10.4/10.5	0.46
	10.5/10.6	0.48
	10.6/10.7	0.41
	10.7/10.8	0.36
	10.9/10.9	0.51
	10.9/10.10	0.24
	10.10/10.11	0.34
	10.11/10.12	0.13
	Median	**0.41**

Considering this example, our approach wrongly predicted 4 co-changes between both files (false positives). In total, in a real scenario, models based on contextual information would be wrong in 4 cases considering the 28 commits tested. Comparing our approach with coupling-based approaches, contextual information reduced the number of false alerts by 43% (12 commits).

We inspect the dataset analyzing how much overlap we could find between a pair of files and the number of co-changes over the releases. We would like to check if the pair of files had co-changes localized in two releases or if the co-changes were spread over releases.

This analysis helps us to understand why the values of recall and precision are higher in our approach then the association rules and why we only used the previous release to build the training set to make the prediction, instead of aggregating the full history of changes, since we built one prediction model per pair of files.

We used the Jaccard index to calculate the similarity and diversity of co-change occurrences over the releases. The Jaccard index is defined as the size of the intersection divided by the size of the union of the sample sets (Pang-Ning et al., 2006). Each sample in our study is the number of co-changes that occurred in one release. In this sense, we considered as intersection if a specific pair of files had at least one co-change in a release (e.g. release N) and one co-change in the consecutive one (e.g. release N + 1).

In Table 8 we note that the similarity for Apache CXF ranges from 0.38 to 0.75, and for Apache Derby it ranges from 0.13 to 0.60. Observing the median values calculated over releases we find more overlap in Apache CXF than Apache Derby. In Apache CXF, the values of similarity are higher in the beginning of the releases analyzed, decrease in the middle, and then start to increase again. In Apache Derby, the similarity values obtained show that pairs of files changed together in the first releases.

The results suggest that co-changes in CXF are more prone to propagate to the next release than in Apache Derby, however, this overlap in less than 63%. As we selected for this work only the top 25 pairs of files with relevant rules, we can note that even considering the best rules obtained by association rules, the co-changes cannot hold the assumption associated with the baseline approach.

Fig. 3 presents the heat map showing the overlap of co-changes between pairs of files. In the Heat map, the more red the color, the more co-changes occurred between the pair of files. The green color means that the pair of files did not change together in a release. The yellow color indicates numbers between 2 and 20. Orange indicates numbers between 21 and 80. The red color indicates numbers above 80.

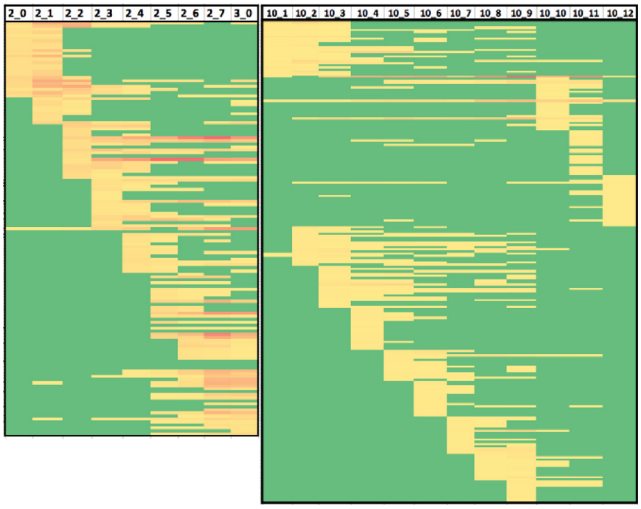

Fig. 3. Heat map of overlapping co-changes between pairs of files over releases.

We found that both projects have similar co-change overlaps between the releases. The values for Jaccard similarity were very close and the Fig. 3 shows a "stairs pattern". It means that a pair of files co-changed, in general, in two or three releases. Because of that we can note a few "line bursts" of yellow, orange or red lines. This pattern shows that it is very hard to find relevant contextual information through the history, because the artifacts analyzed changed in a specific period of software evolution.

> Our classifiers outperform the association rules in both projects considering the precision values. Consequently, fewer false positives were obtained by models based on contextual information.

4.1.2. Does the contextual information perform better than the random classifier?

The AUC metric was designed such that a random classifier would achieve an AUC of 0.5. However, some authors consider that the AUC and accuracy can give a misleading picture of the model performance, especially in case of imbalanced data learning (Powers, 2012), (Powers, 2011). In these cases, MCC is recommended as a better option to evaluate the results (Baldi et al., 2000).

In our case, the imbalance values can be observed in Table 6. We computed the ratio of imbalance by dividing the total instances by the total number of times that Left and Right Files changed together. For Apache CXF, we found 1166 instances where Left File and Right File changed together and 2238 instances where the Left File changed without the Right File. The ratio of imbalance for CXF was 35.29%.

Considering Apache Derby, the ratio was 27.90%, since 1604 instances represented changes for Left and Right file together and 4145 instances occurred when Left File changed without the Right File.

Since the AUC metric is much more commonly used to report results, we report values of AUC and MCC against each other in a bean plot in Figs. 4 and 5. Bean plots are boxplots in which the ver-

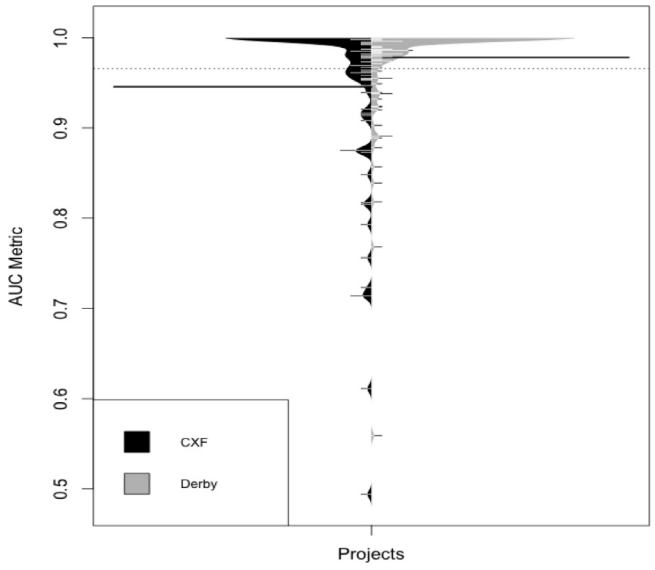

Fig. 4. The AUC values for all classifiers built to predict co-changes over the releases of Apache CXF and Derby.

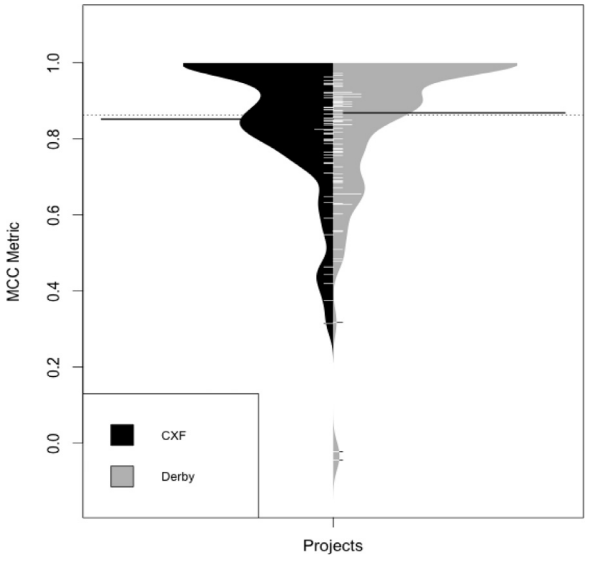

Fig. 5. The MCC values for all classifiers built to predict co-changes over the releases of Apache CXF and Derby.

tical curves summarize the distributions of different datasets. The horizontal black lines indicate the median values (Kampstra, 2008).

We can observe that both projects presented median values for AUC of around 0.95. The other important result is that most classifiers are much better than the random classifier (AUC = 0.5).

Inspecting the results, we note that only seven classifiers obtained AUC below 0.8 for Apache CXF. The values were 0.723 (rank 3, release 2.0), 0.756 (rank 7, release 2.1), 0.494 (rank 7, release 2.3), 0.611 (rank 8, release 2.2), 0.714 (rank 13, release 2.1), 0.714 (rank 18, release 2.6) and 0.793 (rank 23, release 2.5). For Apache Derby, only two models generated AUC results below 0.80. The values were 0.559 (rank 11, release 10.3) and 0.768 (rank 22, release 10.2). The ranks listed in brackets are related to the rule position in our TOP 25 relevant rules.

In Fig. 5, we observe that the values of MCC were very consistent, since the median of the MCC values obtained during the predictions was around 0.85 for both projects. We consider as high correlation all those models that obtained MCC of at least 0.8 between the observed co-changes and the prediction results. Considering Apache CXF, 49 out of 68 classifiers had high correlation. For Apache Derby 93 out of 123 classifiers obtained high correlation.

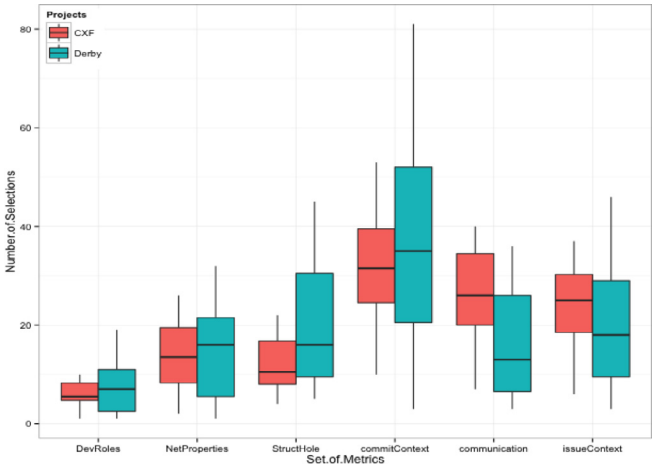

Fig. 6. Box plot comparing the number of selections for each dimension (set of metrics) for each project.

Our classifiers yielded good results for AUC and MCC (above 0.85). We also obtained less than 10.25% false positives and 7.16% false negatives, reducing possible issues for adoption of our approach in practice.

4.2. (RQ2) What are the most important kinds of contextual information when predicting co-changes?

Approach. To study the most important co-change characteristics in our random forest classifiers, we computed the variable importance score for each studied contextual information. The generic function *varImp* was used to characterize the general effect of predictors on the model. The larger the score obtained, the greater is the importance of the co-change contextual information.

For each model we logged the metrics selected. To summarize the importance of each set of metrics we report a boxplot comparing each dimension in Fig. 6. Table 9 presents an aggregated number for each metric selected by release. The red column indicates the metric most selected for each dimension. We also aggregated the total by set of metrics in line "Dim".

Results. Commit Context was an important explanatory factor of co-change prediction for both projects. The other dimensions presented different explanatory power for each project. The number of changed lines (commit context), issue reporter (issue context), number of words (communication context) and closeness (developers' roles) were the most selected.

To compare the number of selections among the dimensions (sets of metrics), we used the Kruskal-Wallis rank sum test: a non-parametric inferential test that compares the distribution of three or more unmatched groups under the null hypothesis that the location parameters of the distribution are the same within each group.

After running the test, using a 0.05 significance level, we concluded that for both projects the number of selections for each dimension followed a non-identical distribution. For Apache CXF the p-value turned out to be 0.0002853 (chi-squared 23.3821, df = 5). Apache Derby presented a p-value of 0.006985 (chi-square = 15.9511, df = 5).

Table 9
Analysis of contextual information importance by feature selection analysis.

Rel. Ver.	Set of Metrics grouped by software dimension (Table IV)																						
	IC				CC				DRC		SH				NP				ComC				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
2.0	13	4	12	1	6	8	3	16	7	1	4	5	5	2	4	3	3	3	11	9	11	1	5
2.1	10	4	9	1	4	5	3	11	5	1	2	2	3	2	3	2	2	2	8	8	7	3	3
2.2	4	1	1	0	2	1	1	3	1	0	1	2	0	1	0	0	1	1	4	2	2	0	3
2.3	21	10	5	1	6	8	7	19	5	0	3	5	2	2	5	3	5	6	19	14	14	2	4
2.4	13	11	5	2	9	10	7	13	8	2	4	5	5	5	7	5	6	8	14	13	13	3	4
2.5	10	6	3	2	5	8	4	10	6	3	5	6	6	5	5	6	6	4	11	9	6	3	3
2.6	12	7	4	3	3	5	7	10	4	1	1	3	2	3	3	2	3	6	11	10	9	0	1
2.7	5	3	2	1	2	3	3	3	4	0	1	1	2	1	1	2	1	2	2	4	4	0	0
Tot	88	46	41	11	37	48	35	85	40	8	21	29	25	21	28	23	27	32	80	69	66	12	23
Dim	**186**				**205**				**48**		**96**				**110**				**220**				
10.1	16	4	17	9	4	14	8	10	13	6	11	8	14	12	9	7	9	7	25	24	18	4	10
10.2	7	3	6	2	1	3	3	3	3	1	3	4	4	5	3	3	5	3	7	6	7	2	6
10.3	20	4	12	4	9	8	11	8	6	5	8	7	9	11	5	3	9	7	17	12	12	3	9
10.4	5	1	6	1	3	2	2	3	3	0	3	2	4	4	3	1	3	1	7	6	7	1	9
10.5	10	1	13	2	6	8	6	5	8	2	9	7	5	4	10	5	6	6	18	11	9	3	10
10.6	6	0	9	2	3	3	3	4	6	1	6	4	2	3	6	2	6	5	10	9	7	1	8
10.7	1	0	2	1	1	1	0	1	1	0	1	2	1	2	0	0	0	1	5	3	2	0	3
10.8	16	1	12	2	5	7	3	12	7	5	8	8	7	10	5	2	7	2	23	13	11	2	15
10.9	14	2	9	2	6	6	3	8	5	6	7	8	6	7	7	3	6	3	16	8	7	1	9
10.10	4	0	2	0	1	1	0	1	1	1	2	2	0	1	1	0	2	0	5	2	1	0	1
10.11	2	0	1	0	1	1	0	1	1	1	2	2	1	1	1	0	1	0	2	0	0	0	1
Tot	101	16	89	25	40	54	39	56	54	28	60	54	53	60	50	26	54	35	135	94	81	17	81
Dim	**231**				**189**				**82**		**227**				**165**				**408**				

Rel Ver. – Release Versions, Tot - Total, Dim - Sum by dimension, Issue Context (IC), Communication Context (CC), Developer Role Communication (DRC), Structural Role (SH), Network Properties (NP), Commit Context (ComC), (1) Reporter, (2) Issue Type, (3) Assignee, (4) is Issue Reopened? (5) Developer Commenters (6) Comments, (7) Commenters, (8) Wordiness, (9) Closeness, (10) Betweenness (11) Efficiency (12) Effective Size (13) Hierarchy (14) Constraint (15) Density (16) Diameter (17) Ties (18) Size (19) Changed Lines (20) Added Lines (21) Deleted Lines (22) Same Ownership (23) Committer.

Fig. 6 shows a boxplot comparing both projects and the number of selections for each set of metrics. We can observe that commit metadata has a mean higher than 30 considering all releases of each project. On the other hand, Developer Role got fewer selections. Since we found differences using a Kruskall-Wallis test, we performed a *posthoc* test using Mann-Whitney-Wilcoxon to compare groups against each other and check if the difference in number of selections was statistically different.

The results showed that, for Apache CXF, there were significant differences between Commit Context and Structural Hole (p-value < 0.006759), between Issue Context and Developers Roles Context (p-value < 0.007501), and between Network Properties and Developers Roles (p-value < 0.0498). We did not find differences between Commit Context and Issue and Communication Context. We also did not find differences between Commit, Communication and Issue Context.

The same happened when we compared Network Properties against the Structural Hole. Based on this comparison, we could notice that Commit, Communication, and Issue Context were the most relevant sets of metrics for Apache CXF. In second position: Network Properties and Structural Role. We used as tie breaker the median of each set of metrics.

Considering Apache Derby, we found significant differences between Commit Context and Communication Context (p-value $<$ 0.01033) and between Issue Context and Network Properties Context (p-value < 0.03678). We also found a significant difference between Network Properties and Developers Roles Context (p-value < 0.02006). We did not find differences between Commit Context and Issue Context and Structural Hole.

The same happened when we compared Communication Context against Network properties. After the statistical analysis for Apache Derby, the sets of metrics were sorted and Commit Context, Issue Context and Structural Hole appeared in the first position. Network Properties and Communication Context appeared in second place. We also analyzed each metric individually. We found that the most selected metrics for Commit, Issue, Communication Context and Developers Roles were the same for both projects.

We qualitatively inspected the models built for both projects. Considering all the 144 classifiers built for Apache CXF, 71.53% of the models used metrics from more than one dimension (28.47% used a unique dimension, 21.53% two dimensions, 15.28% three dimensions, 6.94% four dimensions, 9.03% five dimensions and 18.75% all dimensions). For Apache Derby 62.55% of the models used metrics from more than one dimension (37.25% used a unique dimension, 25.51% two dimensions, 11.11% three dimensions, 3.70% four dimensions, 7.00% five dimensions and 15.23% all dimensions).

> More than one dimension was frequently selected by our classifier. This finding shows that co-change prediction is a multi-dimensional phenomenon.

5. Related work

Some artifacts are frequently changed together throughout software development. Ball et al. (1997) introduced the concept of change coupling, which captures the notion of some artifacts frequently changing together during software development. The more two artifacts are committed together (co-changes), the more change-coupled they become (Ball et al., 1997; Ying et al., 2004). Some benefits of change coupling analysis were discussed by D'Ambros and colleagues (2006, 2009; Oliva and Gerosa, 2015b). For example, change couplings reveal relationships not present in the code or in the documentation. Other researchers showed that change couplings affect software quality (Cataldo et al., 2009; D'Ambros et al., 2009; Kirbas et al., 2014).

Based on the idea that artifacts that changed together in the past are prone to change together in the future, researchers leveraged change couplings to predict co-changes. Zimmermann and colleagues (Zimmermann et al., 2005) built an Eclipse plug-in that collects information about source code changes from repositories and warns the developers about probable co-changes. They used association rules to suggest change coupling among files at method and file level. The authors reported precision values around 30%

and recommended that the analysis should be made at the file-level instead of method-level.

Canfora et al. (Canfora et al., 2010) showed that change dependencies detected with the Granger causality test can be combined with association rules to identify co-changes among files. The combination of both techniques can improve the accuracy of the association rules. The authors showed that the combination of Granger and association rules can lead to prediction models with F-measure values around 30%.

Impact Miner (Dit et al., 2014) was proposed to enable developers to extract co-changes from SVN repositories at method level granularity. The tool was designed to help developers during feature location, based on different types of queries combining different techniques (Information Retrieval, dynamic analysis of source code and repository mining), based on the available sources of information.

Zhou et al. (Zhou et al., 2008) proposed a model to predict co-changes using Bayesian networks. They extracted features like static source code dependencies, past co-change frequency, age of change, author information, change request, and change candidate. They conducted experiments on two open source projects (Azureus and ArgoUML) and reported precision values around 60% and recall values of 40%.

In general, we found similar results for precision when compared to the studies reported, however it is hard to compare works because datasets are different, presenting different techniques and co-changes analyzed. Because of this difficulty to compare we used the association rules as baseline model. Comparing with this approach, widely used in the literature, we obtained better results in terms of recall and F-measure.

Our paper also differs from previous work in that we are considering contextual information that had not been used before. Using information related to developers' communication and issue context is new and presented promising results to reduce the number of false positives and negatives of the previous approaches.

6. Discussion

6.1. Advantages and disvantagens of contextual information models

Zimmermann and colleagues (Zimmermann et al., 2005) reported that it is very difficult to have precise and a large number of recommendations. Models may make many suggestions and issue many false alarms, or make fewer suggestions and be more precise. In this sense, we investigate the effectiveness of contextual information to predict co-changes involving files that co-changed often in each project.

The related work presented in the previous section uses commit data and source code, meanwhile, our approach can be more precise to predict co-changes, but has limitations in terms of covering all commits or files. In this sense, there is a trade-off to be decided by software managers and developers when they are choosing an approach to guide them during the software evolution.

In this study for example, by relaxing the values of support and confidence used by the baseline approach, it is possible to receive more recommendations, however penalizing the precision of those recommendations. In this sense, it is important to mention that the effort to collect data is higher in the contextual information models because association rules only depend on the commit data.

Considering the parameters that need to be configured to use models based on contextual information and the baseline models, there is an advantage to use our approach, since we built the prediction models using a random forest algorithm with default configuration without configuring the parameters to obtain the results. Hence, models based on contextual information are simpler because they do not require prior configuration instead of the baseline models which require a previous configuration of support and confidence values. This problem was also pointed out by Zimmermann et al., (2005) who argued that defining the best thresholds of support and confidence is not easy because they depend on the particular project.

An important question that arises from our case study is related to the period when the training and test sets were created. The release period represents the life cycle to achieve an aim, for example, correct critical bugs, refactor some part of the project, or implement new features. It is not clear what the effect of choosing this timeframe was. We conjecture that a release can capture "related context" to build training sets.

6.2. Does co-change context add anything to the prediction power of existing models?

In this work, we investigated different dimensions from software development that we conjecture can improve the co-change prediction since they represent the context that describes the circumstances of each co-change. In the literature, there are results that suggest that each change has a different proposal, and can be influenced by different reasons (Beyer and Noack, 2005; Canfora et al., 2014; Oliva et al., 2013; Shihab et al., 2012; Tao et al., 2012). Recently, we also found that there is less relation than expected between structural coupling and code changes, suggesting that structural coupling is not a good predictor for software changes (Oliva and Gerosa, 2015a).

Different from the state of the art of bug prediction, where models are normally built using information gathered from all files, we decided to build models for each pair of files. Our decision to build models at pair-level instead of file-level is based on the results obtained in the RQ1. We found that pairs of files co-changed in two or three consecutive releases and that co-changes rarely spread over the releases. This lower overlap between releases and co-changes corroborates our assumption that the recent context is useful to predict co-changes in the consecutive releases, and it is not necessary to aggregate longer periods of history to build models.

In the RQ2, we found that more than one kind of context tends to be used to build the models (dimensions and metrics from Table 1). This result suggests that better models involve more than one source of information, and this is different from the literature, since previous work in this area was based on coupling, using source code or the version control system as source to predict co-changes.

A challenge that arises from our results is how we can integrate, preprocess, and use different sources of information at a large scale to build co-change prediction models. We described these steps in Section 3, but they were designed for the Apache Community and probably need to be adapted to other communities or projects.

As we also noticed in RQ2, different sets of contextual information were selected by each co-change prediction model. Since we observed that commit meta-data was the best dimension, we built models using only contextual information from this dimension to compare the results with the best subset of metrics selected for each co-change model.

Analyzing the results for CXF, using only commit metadata contextual information listed in Table 1, we obtained recall values of 85%, and precision values of 75% (F1 value of 79%). We found that recall values were almost the same, but the precision was worse than using the best metrics. In practical terms it means that using only 4 metrics (2 metrics less than the average) we obtained similar values of recall but the models used increase the number of false positives by 15%. For the Derby project, we obtained recall values of 79%, precision values of 78% (F1 value of 79%). To derby

project we observed that recall and precision values decreased by around 10%.

In practical terms, these results suggest that there is a tradeoff between collecting only one dimension of contextual information or using feature selection analysis after collects all contextual information proposed in this work. In terms of recommendations, at least, using only the commit metadata after 100 recommendations given, the models will return at least 15 wrong recommendations. However, the effort to collect the contextual information and run the models would be simplified.

6.3. Practical aspects of contextual prediction models

We sent messages to mailing lists of both projects to report the results obtained and obtain feedback from each community. Four developers (2 from each project) answered the e-mail.

Recent studies from our research group (Pinto et al., 2016; Steinmacher et al., 2016) found that newcomers need to overcome many barriers to push their first contribution to the community. For example, setting up the environment, debugging the code, and finding useful documentation are examples of these barriers. Besides that, software communities have been receiving a lot of contributions of casual contributors that give one or two contributions and then abandon the project. In this case, we believe that our approach can help these kinds of developers to navigate the code, or to find undeclared (hidden) association in the source code.

For example, a Derby developer said: *"this tool [approach] could be useful in tracking down methods whose switch statements need to be updated when, say, you add a new enum value. In general, this tool could be useful wherever you have undeclared dependencies among files and components, which the compiler can't track."*

According to developers from CXF and Derby, they were surprised by the results because usually to change artifacts to complete an issue, they start debugging the source code or think about some architectural aspect that could help during the task. Developers also pointed out that newcomers have difficulty to navigate the existing code because they are not familiar with the software architecture. In this sense, developers from both communities suggested that the proposed approach could help newcomers to navigate through the source code to complete tasks.

A developer from CXF told us "*well, the long-term contributors usually know how the files are connected but I agree with Christian it might help the newcomers navigate via a project, etc*."

In addition, developers from CXF and Derby suggested that the proposed approach could be implemented as an IDE plugin or like a maven tool that after the first commit the developers could receive suggestions.

"Think the idea of giving people some hint that classes are connected (according to your rules) makes a lot of sense. Like you said this can help newcomers to navigate around the code. For this purpose, an IDE plugin makes sense."

On the other hand, developers disagreed that the approach could help during the code review, as experienced developers know which files change together in a task. This statement can be perceived by a comment we received from the CXF project "*I am not sure about the review part. As a reviewer you will always see the change set and you know you got to review all changes. Your tool could report a file that was expected to be also changed but was not. As this would probably mean that something was forgotten to change a test should catch that."*

However, a developer from Derby pointed out a different perspective concerning the practical aspects of the proposed approach "*…also through code review, running tests, and messages from the compiler. Is your idea that, given a database of change history as you have described it, some tool would be able to notice when the developer makes a certain type of change, and then suggest other related changes that are typically made at the same time? I think that's a pretty interesting idea."*

7. Threats to validity

The first concern is generalizability. In our analysis, we presented two case studies. However, based on this limited scope, our results might not generalize to other projects and domains. To reduce these threats, we used several releases to evaluate our results. However, again, projects with other community organization and process of development can present different results. The choice of two projects and the evaluation made considering the co-changes identified by relevant association rules allowed us to better control and understand the data being analyzed. Replication of this work in a large set of systems is required in order to arrive at conclusions that are more general.

Set of metrics: The set of metrics used might not be complete. We dealt with this threat by performing a selection of measurements along different dimensions of software development. We chose metrics from contextual information related to issue, communication and commit metadata. To select the set of metrics, we were inspired by previous work on prediction models (Hall et al., 2012; Wiese et al., 2014a).

We collected the metrics to build the training and test sets when the issues were in the status of closed/fixed. In this sense, some test sets could have been associated to communication metrics that happened after the commit. We inspected the test set to evaluate the effects of this threat and found that only 351 of 2238 comments to CXF and 1279 of 19,109 comments to Derby were made in this situation. To avoid this threat, we are building a tool to collect data from different sources and generate the test set in the moment that each commit occurs in the repository. Thus, the developer can receive the recommendations immediately after their commit.

Co-changes: A threat related to co-change identification are tangled commits (Herzig and Zeller, 2013). This term is related to the interaction between version control systems and developers. Developers often commit unrelated or loosely related code changes in a single transaction. In our study, this threat is limited as we are grouping commits per issue. In addition, we performed a careful selection of issues, using issues that were closed, fixed and for which the source code was integrated. Considering the Power Law distribution in software engineering analysis, our co-changes files comprise the most important files in that moment of development and contain files with a large number of change couplings between files (see Table 2, *support value*).

Linking issues to commits: As we mentioned in Section 3.2 (DE Step 2), Apache projects usually adopt the pattern [ProjectName-IssueNumber] in the commit message to link commits to issues. With this heuristic, we were able to link 8975 out of 20,288 (44.2%) commits in Apache CXF. In the Apache Derby project, the heuristic performed much better, as we could link 8386 out of 10,701 (78.4%) commits. Discovering issue ID in commit messages to link commits to issues is a heuristic that has been applied several times in the literature (D'Ambros et al., 2012; McIntosh et al., 2014; Śliwerski et al., 2005). On the other hand, Wu et al., (2011) showed that, for many projects, the links between issues and commits are not explicitly reported in the commit message nor in the issue description. The development of more sophisticated linking mechanisms is a current research subject (Wu et al. (2011) and Le et al. (2015)).

Overfitting: Overfitting occurs when a prediction model has random error or noise instead of an underlying relationship. It is important to mention that our models were planned to be more "fitted" to each co-change. In this sense, our classifiers are built for each co-change and rely on a specific subset of the contextual

information. To treat the overfitting of our classifiers, we are applying feature selection analysis to reduce the amount of contextual information used in each model. We do not use a cut-off threshold to select the metrics. The RFE algorithm made the selection based on the performance of each model built to predict the co-changes. It is important to highlight that on average 6 out of 23 metrics were necessary to build good classifiers.

8. Conclusions and future work

Change artifacts are a central aspect of software evolution and maintenance. When developers modify software entities, they must ensure that other entities in the software system are updated to be consistent with these new changes. Co-change prediction approaches aim to assist developers in understanding their software and determining the extent of the propagation of each change, identifying artifacts that are more likely to change together.

Hence, in this paper, we set out to answer this central question: *Can contextual information extracted from issues, developers' communication, and commit metadata improve the accuracy of co-change prediction?*

Through a study of two important Apache projects, we found that the answer is:

RQ1. Our classifiers were able to predict co-changes achieving an AUC of 0.90 on average for both projects and outperforming the association rules considering the precision values. We also obtained fewer false positives (10.25%) and false negatives (7.16%) reducing barriers to a possible adoption of our approach in practice.

RQ2. We observed that co-change is a multidimensional phenomenon. Our classifiers derive much of their explanatory power from commit metadata, as well as issue and communication context. We found that at least 62.75% of the classifiers used metrics from more than one dimension to predict the co-changes.

Limitation. Our classifiers inherently depend on traces of contextual information obtained from the history of software changes. In different scenarios, for example, where the developers are working on a new module in a project, we recommend the use of static/dynamic analysis to predict co-changes.

Future work. To use our classifiers, developers need to know the first artifact they need to change. This action is known as "feature location analysis", but is beyond the scope of this paper. We plan to combine our approach with feature location analysis. As future work, we plan to extend the analysis to other pairs of files, and compare our classifiers with other heuristics and co-change prediction approaches.

Acknowldgment

We thank Fundação Araucária, NAPSOL, NAWEB, FAPESP, and CNPQ (461101/2014-9) for the financial support. Marco Aurelio Gerosa receives individual grants from CNPQ. Igor receives grants from CAPES (BEX 2039-13-3). Gustavo receives an individual grant from CAPES (250071/2013-4).

References

Baldi, P., Brunak, S., Chauvin, Y., Andersen, C.A., Nielsen, H., 2000. Assessing the accuracy of prediction algorithms for classification: an overview. Bioinformatics 16, 412–424.

Ball, T., Kim, J., Siy, H.P., 1997. If your version control system could talk. ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering.

Bavota, G., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A., 2013. An empirical study on the developers' perception of software coupling. In: Proceedings of the International Conference on Software Engineering, pp. 692–701.

Bettenburg, N., Hassan, A.E., 2012. Studying the impact of social interactions on software quality. Empirical Software Engineering.

Beyer, D., Noack, A., 2005. Clustering software artifacts based on frequent common changes. In: Proceedings – IEEE Workshop on Program Comprehension, pp. 259–268.

Bicer, S., Bener, A.B., Caglayan, B., 2011. Defect prediction using social network analysis on issue repositories. In: 2011 International Conference on Software and Systems Process, ICSSP 2011, Co-Located with ICSE 2011, May 21, 2011–May 22, 2011, pp. 63–71.

Bird, C., Nagappan, N., Murphy, B., Gall, H., Devanbu, P., 2009. Putting it all together: Using socio-technical networks to predict failures. In: Proceedings – International Symposium on Software Reliability Engineering. ISSRE, pp. 109–119.

Bird, C., Nagappan, N., Murphy, B., Gall, H., Devanbu, P., 2011. Don'T touch my code!: examining the effects of ownership on software quality. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 4–14.

Bohner, S.A., Arnold, R.S. (Eds.), 1996, Software Change Impact Analysis. Los Alamitos, California, USA, IEEE Computer Society Press.

Breiman, L., 2001. Random forests. Mach. Learn. 45, 5–32.

Brézillon, P., Borges, M.R., Pino, J.A., Pomerol, J., 2004. Context-awareness in group work: three case studies. In: International Conference on Decision Support Systems, pp. 115–124.

Briand, L.C., Wust, J., Lounis, H., 1999. Using coupling measurement for impact analysis in object-orientedsystems. In: Proc. IEEE Int. Conf. Softw. Maint. – 1999 (ICSM'99). 'Software Maint. Bus. Chang.

Canfora, G., Ceccarelli, M., Cerulo, L., Di Penta, M., 2010. Using multivariate time series and association rules to detect logical change coupling: an empirical study. IEEE International Conference on Software Maintenance. ICSM.

Canfora, G., Cerulo, L., Cimitile, M., Di Penta, M., 2014. How changes affect software entropy: an empirical study. Empir. Softw. Eng. 19, 1–38.

Cataldo, M., Mockus, A., Roberts, J.A., Herbsleb, J.D., 2009. Software dependencies, work dependencies, and their impact on failures. IEEE Trans. Softw. Eng. 35, 864–878.

Ceccarelli, M., Cerulo, L., Canfora, G., Penta, M.Di, 2010. An eclectic approach for change impact analysis. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, 2, pp. 163–166.

Conway, M.E., 1968. How do committees invent. Datamation 14, 28–31.

D'Ambros, M., Lanza, M., Lungu, M., 2006. The evolution radar: visualizing integrated logical coupling information. In: MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories, pp. 26–32.

D'Ambros, M., Lanza, M., Robbes, R., 2009. On the relationship between change coupling and software defects. In: Reverse Engineering, 2009. WCRE '09. 16th Working Conference on, pp. 135–144.

D'Ambros, M., Lanza, M., Robbes, R., 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. In: Empirical Software Engineering, pp. 531–577.

De Santana, A.M., Da Silva, F.Q.B., De Miranda, R.C.G., Mascaro, A.A., Gouveia, T.B., Monteiro, C.V.F., Santos, A.L.M., 2013. Relationships between communication structure and software architecture: an empirical investigation of the Conway's Law at the Federal University of Pernambuco. In: Proceedings – 2013 3rd International Workshop on Replication in Empirical Software Engineering Research, RESER 2013, pp. 34–42.

De Souza, C.R., Quirk, S., Trainer, E., Redmiles, D.F., 2007. Supporting collaborative software development through the visualization of socio-technical dependencies. In: 2007 International ACM Conference on Supporting Group Work, GROUP'07, November 4, 2007–November 7, 2007, pp. 147–156.

Dey, A.K., Abowd, G.D., Salber, D., 2001. A conceptual framework and a toolkit for supporting the rapid prototyping of context- aware applications. Hum. – Comput. Interact. 16, 97–166.

Dit, B., Wagner, M., Wen, S., Wang, W., Linares-Vásquez, M.., Poshyvanyk, D., Kagdi, H., 2014. ImpactMiner: a tool for change impact analysis. In: 36th International Conference on Software Engineering, ICSE Companion 2014 – Proceedings, pp. 540–543.

Gall, H., Hajek, K., Jazayeri, M., 1998. Detection of logical coupling based on product release history. In: Proceedings. Int. Conf. Softw. Maint.

Gethers, M., Dit, B., Kagdi, H., Poshyvanyk, D., 2012. Integrated impact analysis for managing software changes. In: Proceedings - International Conference on Software Engineering, pp. 430–440.

Gethers, M., Poshyvanyk, D., 2010. Using relational topic models to capture coupling among classes in object-oriented software systems. IEEE International Conference on Software Maintenance. ICSM.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2012. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng 38 (6), 1276–1304.

Hassan, A.E., Holt, R.C., 2004. Predicting change propagation in software systems. In: IEEE International Conference on Software Maintenance. ICSM, pp. 284–293.

Herzig, K., Zeller, A., 2013. The impact of tangled code changes. In: IEEE International Working Conference on Mining Software Repositories, pp. 121–130.

Kagdi, H., Gethers, M., Poshyvanyk, D., 2013. Integrating conceptual and logical couplings for change impact analysis in software. Empir. Softw. Eng. 18, 933–969.

Kampstra, P., 2008. Beanplot: a boxplot alternative for visual comparison of distributions. J. Stat. Softw. 28, 1–9.

Kirbas, S., Sen, A., Caglayan, B., Bener, A., Mahmutogullari, R., 2014. The effect of evolutionary coupling on software defects: an industrial case study on a legacy system. In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14. New York, NY, USA. ACM, pp. 6:1–6:7.

Kuhn, M., 2008. Building predictive models in R using the caret package. J. Stat. Softw. 28, 1–26.

Kwan, I., Cataldo, M., Damian, D., 2012. Conway's law revisited: the evidence for a task-based perspective. IEEE Softw..

Le, T.D.B., Linares-Vásquez, M., Lo, D., Poshyvanyk, D., 2015. RCLinker: automated linking of issue reports and commits leveraging rich contextual information. In: IEEE International Conference on Program Comprehension, pp. 36–47.

Malik, H., Hassan, A.E., 2008. Supporting software evolution using adaptive change propagation heuristics. In: Software Maintenance, 2008. ICSM 2008. IEEE International Conference on, pp. 177–186.

McIntosh, S., Adams, B., Nagappan, M., Hassan, A.E., 2014. Mining co-change information to understand when build changes are necessary. In: Proc. of the 30th Int'l Conf. on Software Maintenance and Evolution (ICSME), pp. 241–250.

Meneely, A., Williams, L., Snipes, W., Osborne, J., 2008. Predicting failures with developer networks and social network analysis. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering – SIGSOFT '08/FSE-16, p. 13.

Mockus, A., 2010. Organizational volatility and its effects on software. In: Proceedings of the 2010 Foundations of Software Engineering Conference, pp. 117–126.

Moser, R., Pedrycz, W., Succi, G., 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. 2008 ACM/IEEE 30th Int. Conf. Softw. Eng..

Oliva, G.A., Gerosa, M.A., 2015a. Experience report: how do structural dependencies influence change propagation? an empirical study. Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015.

Oliva, G.A., Gerosa, M.A., 2015b. Change coupling between software artifacts: learning from past changes. In: Bird, C., Menzies, T., Zimmermann, T. (Eds.), The Art and Science of Analyzing Software Data. Morgan Kaufmann, pp. 285–324.

Oliva, G.A., Steinmacher, I., Wiese, I., Gerosa, M.A., 2013. What can commit metadata tell us about design degradation? In: Proceedings of the 2013 International Workshop on Principles of Software Evolution – IWPSE 2013. ACM Press, p. 18.

Orso, A., Apiwattanapong, T., Law, J., Rothermel, G., Harrold, M.J., 2004. An empirical comparison of dynamic impact analysis algorithms. In: Proceedings. 26th Int. Conf. Softw. Eng..

Pang-Ning, T., Steinbach, M., Kumar, V., 2006. Introduction to data mining. Libr. Congr. 796.

Pinto, G.H., Steinmacher, I., Gerosa, M.A., 2016. More common than you think: an in-depth study of casual contributors. In: 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'2016), pp. 1–12.

Powers, D.M., 2011. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. J. Mach. Learn. Technol. 2, 37–63.

Powers, D.M.W., 2012. The problem of area under the curve. In: Proceedings of 2012 IEEE International Conference on Information Science and Technology, ICIST 2012, pp. 567–573.

Leano, R., Kasi, B.K., Sarma, A., 2014. Recommending task context: automation meets crowd. International Workshop on Context in Software Development (FSE Companion).

Radjenović, D., Heričko, M., Torkar, R., Živkovič, A., 2013. Software fault prediction metrics: a systematic literature review. Inf. Softw. Technol. 55 (8), 1397–1418.

Rahman, F., Devanbu, P., 2013. How, and why, process metrics are better. In: Proceedings – International Conference on Software Engineering, pp. 432–441.

Revelle, M., Gethers, M., Poshyvanyk, D., 2011. Using structural and textual information to capture feature coupling in object-oriented software. Empir. Softw. Eng. 16, 773–811.

Schröter, A., Aranda, J., Damian, D., 2012. To talk or not to talk: factors that influence communication around changesets. In: Proc. ACM 2012 Conf. Comput. Support. Coop. Work - CSCW '12, pp. 1317–1326.

Shihab, E., Hassan, A.E., Adams, B., Jiang, Z.M., 2012. An industrial study on the risk of software changes. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pp. 62:1–62:11.

Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W.M., Ohira, M., Adams, B., Hassan, A.E., Matsumoto, K.I., 2010. Predicting re-opened bugs: a case study on the Eclipse project. In: Proceedings – Working Conference on Reverse Engineering, WCRE, pp. 249–258.

Steinmacher, I., Treude, C., Conte, T., Gerosa, M.A., 2016. Overcoming open source project entry barriers with a portal for newcomers. In: 38th International Conference on Software Engineering, pp. 1–12.

Tao, Y., Dang, Y., Xie, T., Zhang, D., Kim, S., 2012. How do software engineers understand code changes?: an exploratory study in industry. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pp. 51:1–51:11.

Tsay, J., Dabbish, L., Herbsleb, J., 2014. Influence of social and technical factors for evaluating contribution in GitHub. In: Proceedings of the 36th International Conference on Software Engineering, pp. 356–366.

Wasserman, S., Faust, K., 1994. Social network analysis: Methods and Applications. Cambridge University Press.

Wiese, I.S., Côgo, F.R., Ré, R., Steinmacher, I., Gerosa, M.A., 2014a. Social metrics included in prediction models on software engineering: a mapping study. In: Wagner, S., Penta, M.Di (Eds.), The 10th International Conference on Predictive Models in Software Engineering, {PROMISE} '14. Torino, Italy, September 17, 2014. ACM, pp. 72–81.

Wiese, I.S., Junior, D.N., Ré, R., Steinmacher, I., Gerosa, M.A., 2014b. Comparing communication and development networks for predicting file change proneness: An exploratory study considering process and social metrics. ECEASST 65.

Wiese, I.S., Kuroda, R.T., Junior, D.N.R., Ré, R., Oliva, G.A., Gerosa, M.A., 2014c. Using structural holes metrics from communication networks to predict change dependencies. In: Baloian, N., Burstein, F., Ogata, H., Santoro, F.M., Zurita, G. (Eds.), Collaboration and Technology – 20th International Conference, {CRIWG} 2014, Santiago, Chile, September 7-10, 2014. Proceedings, Lecture Notes in Computer Science. Springer, pp. 294–310.

Wolf, T., Schröter, A., Damian, D., Nguyen, T., 2009a. Predicting build failures using social network analysis on developer communication. In: Proceedings – International Conference on Software Engineering, pp. 1–11.

Wolf, T., Schröter, A., Damian, D., Panjer, L.D., Nguyen, T.H.D., 2009b. Mining task-based social networks to explore collaboration in software teams. IEEE Softw. 26, 58–66.

Wu, R., Zhang, H., Kim, S., Cheung, S.C., 2011. ReLink: recovering links between bugs and changes. Proc. ACM SIGSOFT Symp. Found. Softw. Eng. 15–25.

Ying, A.T.T., Murphy, G.C., Ng, R., Chu-Carroll, M.C., 2004. Predicting source code changes by mining change history. IEEE Trans. Softw. Eng. 30, 574–586.

Zanetti, M.S., 2012. The co-evolution of socio-technical structures in sustainable software development: lessons from the open source software communities. In: Proceedings - International Conference on Software Engineering, pp. 1587–1590.

Zhang, H., Gong, L., Versteeg, S., 2013. Predicting bug-fixing time: an empirical study of commercial software projects. In: Software Engineering (ICSE), 2013 35th International Conference on, pp. 1042–1051.

Zhou, Y., Wursch, M., Giger, E., Gall, H., Lu, J., 2008. A Bayesian network based approach for change coupling prediction. In: Fifteenth Work. Conf. Reverse Eng. Proc., pp. 27–36.

Zimmermann, T., Nagappan, N., Guo, P.J., Murphy, B., 2012. Characterizing and predicting which bugs get reopened. In: Proceedings – International Conference on Software Engineering, pp. 1074–1083.

Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A., 2005. Mining version histories to guide software changes. IEEE Trans. Softw. Eng. 31, 429–445.

Igor Scaliante Wiese is a PhD student in Computer Science at the Institute of Mathematics and Statistics at University of São Paulo in 2015. Currently, he is a Professor at the Federal University of Technology - Paraná (UTFPR). His research lies in the intersection between Software Engineering (SE) and Computer Supported Cooperative Work (CSCW). Currently, his focus is on software dependencies analysis, specially to support developers during software changes. His interest is in Open Source Software, Human Aspects of Software Engineering, Empirical Software Engineering, and Mining Software Repositories techniques.

Reginaldo Ré completed a PhD in Computer Science at the Institute of Mathematics and Computing from São Carlos (SP). He is a Professor at the Federal University of Technology – Paraná (UTFPR). Reginaldo has experience in Computer Science, focusing on Software Engineering, acting on the following subjects: frameworks, software tests, mining software repositories and cloud computing.

Igor Steinmacher completed a PhD in Computer Science at the Institute of Mathematics and Statistics at University of São Paulo in 2015. Currently, he is a Professor at the Federal University of Technology - Paraná (UTFPR). His research lies in the intersection between Software Engineering (SE) and Computer Supported Cooperative Work (CSCW). Currently, His interest is in on the support of newcomers to Open Source Software development communities, Human Aspects of Software Engineering, Empirical Software Engineering, and Mining Software Repositories techniques.

Rodrigo Kuroda is a Master Degree student at Federal University of Technology – Paraná (UTFPR). Rodrigo has experience in programming languages, web frameworks, and, mining software repositories. His Master Degree is focused in build tools to support software engineering tasks.

Gustavo Ansaldi Oliva is a PhD Student of Computer Science at the University of São Paulo under the supervision of Marco Aurélio Gerosa. He worked as a software developer at IBM Brazil for more than 3 years. He also worked for HP Labs during 3 months to conceive and implement workflow change impact analysis techniques. Gustavo's expertise in on Software Engineering and related fields. He has published several papers in the topics of software evolution, service-oriented computing, and free/libre open source software. In his PhD, Gustavo is researching better ways to identify logical dependencies from version control systems.

Christoph Treude completed a PhD in Computer Science at the University of Victoria in 2012. After PostDocs in Canada and Brazil, I am now working as a faculty member in the School of Computer Science, University of Adelaide, Australia. His research strives to advance software development by conducting research on the automated analysis of software documentation and recommendation systems for software engineering. In his PhD research, Christoph Treude studied the role of social media artifacts, such as tags or the popular Q&A website Stack Overflow, in collaborative software development.

Marco Aurélio Gerosa is an Associate Professor in the Computer Science Department at the University of São Paulo, Brazil. His research lies in the intersection between Software Engineering and Social Computing, focusing on the fields of empirical software engineering, mining software repositories, software evolution, and social dimensions of software development. He receives productivity fellowship from the Brazilian Council for Scientific and Technological Development. In addition to his research, he also coordinates award-winning open source projects.