Predicting Co-Changed Files: An External, Conceptual Replication

Abstract

A software project can comprise of several, highly connected files. A software developer, who does not know the files that are connected to which are developed or that are changed by another developer, may induce faults by while missing necessary edits on all related files. We build a prediction model for identifying files that should be edited together during a code change, and evaluate the performance of our model on two Apache projects' development history over more than 10 years. We conduct an external, conceptual replication study based on Wiese et al.'s prior work on predicting co-changed files. Our study shares the same goal but differentiates the experimental design in terms of data set construction, selection of file pairs, feature selection and the model output. Our prediction model's results, although the same performance measures are used, are much lower than what is reported in Wiese et al.'s study, mainly due to the differences in calculating these measures. The models evaluated at commit granularity could achieve 20% and 45% lower recall and precision rates, respectively, than those aggregated over all file-pairs. Although it is practically more useful, predicting all files that will be co-changed together during a commit is more challenging than predicting whether a particular file will be changed in that commit. More information about the context of a co-change, the degree of centrality of a file in the project, or project characteristics could reveal more insights in building such predictors in the future.

Keywords: Mining software repositories, association rule mining, predicting co-changed files, replication study.

1. Introduction

Software projects consist of thousands of files among which contextual, functional or other kinds of dependencies exist. A change made on a source file might require several changes on other files due to such dependencies. A developer's misunderstanding of the code or lack of experience might hinder necessary changes that should be done on all relevant files. This eventually raises quality issues in software projects. Empirical studies focusing on co-change prediction aim to preserve the code integrity and reliability by guiding developers to necessary code fragments that need to be inspected or edited with regard to a particular code change.

There are mainly two approaches for developing co-change prediction models in the literature. The first one utilizes dependency graphs of a software project extracted at any entity level, i.e., from dependencies between business requirements, to static call dependencies between methods [1], while the second one makes use of software code repositories and builds learning-based models [2]. The first approach often requires manual work to update change impact set incrementally, and the changes are not semantically related and analysed [1]. The second approach is more dynamic in nature as it is based on automatic data collection from source code and issue repositories over several releases and updating the prediction models accordingly.

In this paper, we analysed a recent work by Wiese et al. [3] that proposes co-change prediction models for source files using contextual information from version control systems and issue repositories, and chose it as our baseline empirical study. We conceptually replicated the baseline in [3] by extracting more than 10 years of development data of two projects, Apache Derby and Apache CXF, and calculating commit and developer related features. We then built co-change prediction models for a subset of files based on their ranking over commits, and associated co-changed files. Our study differs from the original study [3] in terms of experimental protocol in selecting co-changed file pairs, model construction, and performance evaluation.

2. Related Work

One of the early works on tracking file co-changes through mining code repositories is proposed in [4]. The authors in [4] show that historical commit data can be used to track code evolution. Another study by Gall et al. [5] also used the historical release data from version control systems, and they were able to predict evolutionary coupling between files. Later, Zimmermann et al. [6] developed an Eclipse plugin to predict co-change occurrence at method level, again using the information obtained through mining software repositories. The designed tool was able to suggest some co-changes, but its precision was around 30%. The false alarm rates were very high, and further steps were needed to be taken to improve the accuracy of suggestions.

There have been also other works using different approaches in the context of co-change prediction. For instance, Canfora et al. [7] extracted association rules from the historical code changes to predict file co-changes, and obtained 30% F-measure. The reason of having so low F-measure was due to high false alarms that the association rules generated. The rules are successful at finding co-changes but they also have very high false positives. Wiese et al. [3] proved this in their study by comparing association rules with contextual features. Moreover, Hassan et al. [8] investigated change propagation, and used different heuristics to predict co-changes at the entity granularity by analysing five large open source software projects. They defined heuristics according to their data source and pruning techniques. They obtained the best recall performance when using the entity based historical co-change data with hybrid pruning method, which combines both frequency and recency methods. In order to improve precision performance, Hassan et al. [8] used hybrid heuristics and combined the results of entity based historical co-change and entity based code structure using code layout heuristics. With this new approach, precision increased up to 49%, while recall decreased down to 51%. The authors of [8] stated that more sophisticated heuristics should be developed to obtain better performance.

Macho et al. [9] conducted a study to improve the prediction of *build* co-changes by using the detailed information on source code changes and commit categories. They obtained significantly higher performance compared to prior works, and planned to improve the performance more by including the issue tracking system into their prediction model. Furthermore, Kouroshfar [10] investigated in his article whether there is a relationship between co-change dispersion and the software quality. He used four open source Apache projects and showed that the location of co-changes has an impact on the code defects.

2.1 Baseline Study

Wiese et al. [3] conducted an empirical study to observe whether change patterns could be identified using contextual information from both issue tracking and version control systems. They built predictive models using Random Forest classifier that is trained with the previous changes of file pairs, and recommends whether the second file in the pair should be changed when the first file in the pair has changed. One separate model is trained for every file pair, but this was not done for all file pairs that exist in the historical code changes. They grouped all the commits that were done for an issue to a single change transaction, and all the files that are changed by that change transaction are considered as a co-changed file pair. Association rules based on support and confidence values were generated for each co-change file pairs. Wiese et al. [3] selected the top 25 relevant association rules as the list of co-changed file pairs. They later built models for these 25 pairs by extracting features from issues, commit history, and communication network. The prediction models reported in [3] on Apache Derby and Apache CXF projects, achieve around 90% precision and recall values. Although this original study provides remarkable performance in the context of co-change prediction, there might be several reasons to observe such high recall and precision rates.

First and foremost, the classifier building and evaluation process depicted in [3] may not represent the change propagation in reality. For instance, when a developer makes a change on file A, then the prediction model trained on (A,B) pairs is used to recommend if file B should also be changed. But if another file, say C, also needs to be changed, another model trained on (A, C) pair is used for recommendation. However, in practice, when a developer changes file A, he/she wants the list of files that should be changed together with A. So all models having file A in their co-changed file set (e.g. (A, B), (A, C), (A, D)) should provide recommendations at each commit including A.
Second, the performance evaluation method in [3] might have led to such high rates due to their classifier building strategy. The original study reports precision and recall over all models used in multiple releases of two software products, although in each release, there exist multiple models corresponding to several file

pairs. So the recall and precision values do not depict the performance of separate models with different co-changed pairs. Furthermore, performance evaluation was made by considering each model trained on a pair (A, B) [3] although this does not resemble the practical scenario. When a developer changes an entity (A), he/she would like to learn what other entities need to be changed together with A. The number of recommended entities should ideally be compared with the actual changed entities, and recall and precision should be calculated over all recommended entities for a single commit. This way, it is easier to assess which entities are correctly predicted or give false alarms, and discuss how co-changed pairs could be better predicted.

3. Methods and Materials

Our research is defined as an external and conceptual replication of the baseline study published in [3]. In [11], authors discuss all the types of replication studies and divide them into two categories: “internal” and “external”. It defines a study “internal” if the people, who conduct both the original and the replication studies, are the same, and “external” if the two studies are conducted by different researchers. The authors in [11] also state that replications, which use different experimental protocol in order to check the baseline experimental results, are named as “conceptual replication”. Shepperd et al. [12] discuss the importance of the replication studies, and argue that internal replications have been reported more often than the external ones in the area of empirical software engineering. Silva et al. [13] also supports [12] that the replication studies are necessary in any empirical science, and external ones should be encouraged more in order to increase the quality in this field.

We share the same goal, with [3], of recommending co-changed entities using contextual data from version control systems, while our study differs from the baseline study with regard to the features used, classifier building and evaluation strategies that we believe better illustrates the developer’s needs in practice. Wiese et al. [3] identify six dimensions to investigate the performance of each in predicting co-changed files. These dimensions can be collected from an issue tracking system (issue, communication over issues, developer’s role in communication, structural hole in communication, communication properties) and a version control system (commit context). Although they argue that co-change prediction is a multi-dimensional phenomenon, the most frequently selected features in these models are from two dimensions: commit context and developer’s role in communication [3]. However, collecting all these dimensions is very costly, especially for large scale software projects connected to multiple version control and issue tracking systems. Therefore, we reduced our feature set to the metrics from the commit and communication dimensions only, as concluded by [3].

Moreover, Wiese et al. [3] found co-changed file pairs based on issues attached to the commits, and selected top 25 co-changed file pairs based on support and confidence values derived from association rules over all commits. We have revised this approach by considering the top-5 files changed in all commits, because a model built for a file can only be successful if they are enough number of changes associated with the file in commit history. Then we identified pairs with these top-5 files, based on the support and confidence values derived from association rules over all commits. The details for these association rule calculations will be provided in the next subsections.

Finally, we modified the calculation of performance measures from ‘recall and precision over different models for the selected file pairs’ [3] to ‘recall and precision of a single model that recommends multiple entities to a commit’. The details of performance evaluation are further explained in the corresponding subsection. We also report the performance measures achieved using the approach in [3] to compare our findings with the original study.

3.1 Data Used

Wiese et al. [3] analysed two repositories, namely Apache Derby and Apache CXF, to build co-change prediction models. We mined the GitHub repositories of the same two projects to compare and contrast our findings with the original study. While extracting the commit history, we used GitHub API. As the features we selected are from the commit and developer communication contexts, it was enough to mine only the GitHub repos in our study. The descriptive statistics for Apache Derby and CXF are reported in Table 1. Projects have 10-12 years of development data, and thousands of file edited over the years by 17 to 20 developers. Note that GitHub API provides historical change data on the main branch, and hence, we did not consider branches generated from the main branch in our research.

Table 1. Descriptive statistics for the projects used

	Apache Derby	Apache CXF

Time interval of the commits extracted	11 August 2004 - 3 February 2018	23 April 2008 - 23 April 2018
Total no. of commits	8175	13999
Total no. of files	6179	19233
Total no. of co-changed files	17	20

3.2 Calculation of Confidence Values

Wiese et al. [3] prepared train and test data sets for each file pair, which were changed in the same commit group related to an issue, and selected the top 25 with regard to their confidence values. In order to calculate the confidence values, they firstly calculated the support values of each association rule (r), which is a file pair as mentioned before. The support and confidence calculations are as follows:

$$support\ (\mathrm{r}) = support\ (I \to J) \\ = support\big(\{f_i\} \cup \{f_j\}\big)$$

$$confidence(r) = confidence(I \to J)$$

$$confidence(\mathrm{I} \to \mathrm{J}) \ = \frac{support(\mathrm{I} \to \mathrm{J})}{support(I)} = \frac{support(\{f_i\} \cup \{f_j\})}{support(\{f_i\})}$$

The support values refer to the total number of transactions on fi where other files fj are also changed. In other words, it is the number of common transactions. While the support calculation takes into account the file pairs only, and does not take into consideration the order of their association rule, the confidence calculation considers association rules in which the order of the files is important. More specifically, two association rules can be made from one file pair. For example, if the file pair consists of files “a” and “b”, then the possible association rules are a->b and b->a. While their support values are the same since it equals to total common transactions, their confidence values are different because it is a fraction, which is obtained by the total number of common transactions divided by the total number of the transactions of the first file of the association rule. Therefore, selection of the first file matters.

In this study, we obtained the co-changed file pairs’ and calculated their confidence values based on the formulas above. Then, the first file in co-change file pairs (a,b) is determined by choosing the top-5 files that are changed the most, i.e., based on the total number of commits, whereas the second file in a pair (b) is found based on the confidence value between files a and b. If the confidence value between a and b is at least 0,15, we chose these two files as one of the co-changed pairs. This confidence value will represent the ratio of co-changes over all changes during training.

3.3 Feature Extraction

In order to determine which features could be used, the information that the API provides are analysed. Based on this, we decided that the features associated with commit and developer communication context in Wiese et al. [3] could be calculated for our study. These features, as stated above, are found as the most frequently selected ones to predict co-changes in the original study. The list of features and their descriptions are provided below:

Commit Context:

- *Date:* The date, when the commit is done. It is numeric and is determined by calculating the time difference in days between a commit and the first commit date.
- *Developer:* The name and surname of the developer who made the commit.
- *The number of the changed lines of code:* It is the total number of the changed lines of code belonging to the file in the commit.
- *The number of the added lines of code:* It is the total number of the added lines of code belonging to the file in the commit.
- *The number of the deleted lines of code:* It is the total number of the deleted lines belonging to the file in the commit.

Developer Communication Context:

These features are derived from communication matrix in which developers are rows, files are columns, and each index (d,f) in the matrix represents whether the developer d makes changes on file f. It is a matrix of binary values (0,1) indicating developer’s contribution to a file. This matrix is then converted to closeness and betweenness features using UCINET software to represent a developer’s level of communication with others.

- *Closeness:* It is the total number of the distances (edges) between a developer and the other developers based on the number of files edited by the developer. This feature is obtained by extracting the adjacency matrix of the developers, i.e. developer-developer matrix, from the communication matrix.

- *Betweenness:* It is the number of times that a developer appears in the shortest paths between all developers. This feature is also obtained by extracting the adjacency matrix of the developers from the communication matrix.

To assign class labels to the co-changes, we chose the most frequently changed files (as the first file in a pair) and the files co-changed with the selected files (as second file in a pair). Thus, co-changed file pairs are labelled as 1 when both elements of the pair are changed, while they are labelled as 0 when only the first of the pair is changed. Consequently, the data sets consist of the features related to the commit and developers' communication network, and numerical labels, which indicates if there is a co-change or not (0 or 1). Please note that different models will later be used to predict if a co-change occurs for each file pair or not.

3.4 Model Construction

As mentioned earlier, the prediction models are constructed for the file pairs. The first item in the pair is one of the top-5 file list, the second item is the file that changed together with the first, and had at least 0,15 confidence. The training and test datasets of each model associated to a file pair are constructed as follows:

For each co-change file pair (a,b),

- 80% of the commits where both files (a,b) are changed, and 80% of the commits where only first file (a) is changed are added to the training set. The rest, 20%, of the commits are added to the test set.
- Training set contains only co-changed state of (a,b) pair as the labels. As the number of co-changed instances (1) is larger than those of not co-changed (0), we applied under-sampling on the majority class in training set. In other words, the number of not co-changed instances is randomly picked from the original dataset until there exist equal number of instances from both classes (0 and 1).
- Test set contains co-changed state of all pairs corresponding to file *a*, e.g., ((a,b),..(a,x)), as labels. This is done to make predictions on all test instances, using the models constructed for all pairs consecutively.
- The Random Forest algorithm is used to build the model on the training set for all file pairs separately. We chose the same algorithm used in [3] to keep our analysis comparable with the original study's.
- Each model makes predictions on the test instances such that for a file *a*, the models provide *0* or *1* indicating the corresponding file (b,..,x) is likely to be changed with *a*.
- The predictions are assessed per test instance, i.e., recall and precision are calculated based on the number of correctly predicted co-changes over all pairs.

This process is repeated five times for each file pair, and the model is constructed at each iteration using a different training and test set. It should be noted that the commits in two sets must be different, and the test set should not contain duplicate commits of the first file in a pair, coming from multiple model's test sets. To avoid this, when the sets are generated for each file-pair, we check if the training contains any instance from the test set, and if so, we replaced those with new instances by random sampling from the dataset.

The process of model construction is depicted in Fig. 1. Based on Fig.1, suppose there is a file "a" and it is one of top-5 files changed in most of the commits in a dataset. All files that are co-changed with "a" are found first. Then based on the file pairs' confidence values, the file pairs that are relevant to our context are identified. In Fig. 1, these files are named as "b" and "c". First, the training and test sets are generated for (a,b) and (a,c) file pairs following 80/20% rule. Second, while the training sets are kept separately to build different models, the test sets of (a,b) and (a,c) are combined to have one test set. At this stage, we paid attention not to include any commit twice in both sets, because the test set of (a,b) might contain a commit in which file c might also be changed. In such a case, this instance could be in the training set of (a,c). We had to double check these cases before proceeding to model construction. Our final test set corresponds to commits in which file "a" has changed, and all the models would make a prediction about whether b or c should also change in the corresponding commit.

Moreover, the instances of the training sets (a,b) and (a,c) have only one label, which indicates that if b or c has co-changed with a or not, while the instances of the test for file a contain labels for both b and c.

3.5 Performance Evaluation

In order to measure the performance of our prediction models, we calculated confusion matrices and reported precision and recall, which were also used in [3]. Table 2 shows the representation of a confusion matrix for a co-change prediction. While precision gives what percentage of the model's predictions for co-changes are true, recall measures what percentage of the file pairs that are actually co-changed are truly predicted as co-change. One of the differences between our study and Wiese et al. [3] lies in the calculation of these measures.

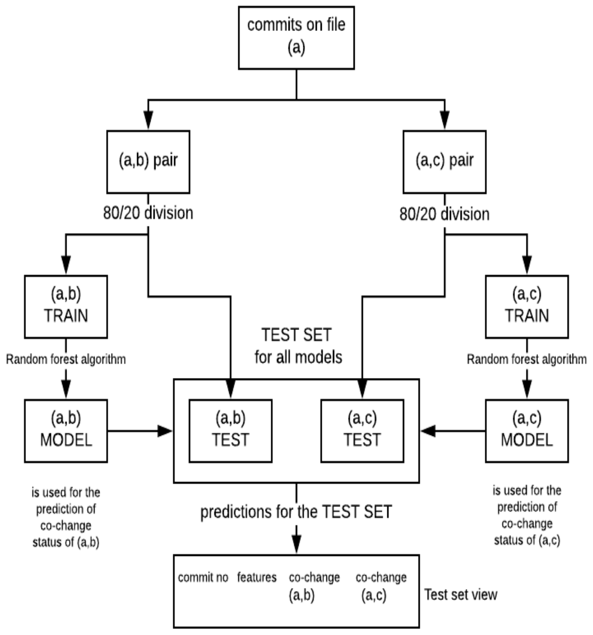

Figure 1. Model construction process used in our study

Table 2. Confusion matrix

		Predicted	
Actual		*0*	*1*
	No co-change (0)	*TN*	*FP*
	Co-change (1)	*FN*	*TP*

Precision: $\frac{\mathbf{TP}}{(\mathbf{TP}+\mathbf{FP})}$ *Recall*: $\frac{\mathbf{TP}}{(\mathbf{TP}+\mathbf{FN})}$

In [3], recall is calculated based on the total number of correctly classified instances over all models. So if two models for (a,b) and (a,c) correctly classify 2/3 and 2/2 co-changes, then recall is 4/5, 80% in [3]. In our study, we calculated recall for each instance in test set of a. Let's assume that in total of these five commits, one commit has all three files changed (a, b, c), and the models correctly classify b and c. So for this commit, recall is 100%. If two of the five commits have only (a,b) changed, and only one is correctly classified, recall is 0% and 100% respectively. Finally, if two of the five commits have only (a,c) changed, and the model correctly classifies one, recall is 0% and 100%. In this scenario, the recall of our model is (0+100+100+100+0)/5 = 60%. We believe such evaluation is practically more useful for developers as they would see how many files are correctly recommended by the model in a commit. On the other hand, our approach produces lower recall and precision rates than [3]. Therefore, we expect to see worse performance than [3]. We report recall and precision rates using both our approach and Wiese et al.'s approach in Results Section.

4. Results

Our results on Apache Derby and CXF projects are presented in Table 3. For each project, models built for the top-5 files and the performance on the test sets of these five files are reported in terms of precision and recall. The average performance is also presented in Table 3 in two different ways: The first average is based on our performance calculation, whereas the second average is based on the approach used in [3]. As shown in the table, the recall values change from file to file; in some cases, we could predict up to 100% of files co-changed with the target file, e.g. file #3 in Apache Derby, whereas in other cases, it is only possible to correctly predict half of the files co-changed with the target file, e.g. file #4 in Derby and file #1 in CXF. On average, 59% to 76% of co-changes can be predicted with the model that we proposed. Precision values, on the other hand, show that around half of our predictions are correct, i.e. false alarms are high. When the performance measures are calculated based on Wiese et al. [3], it is seen that both the precision and recall values have increased up to 92% and 89%, respectively. These rates are in line with what was reported by Wiese et al. [3].

We have also calculated precision and recall values for the five files and their associated pairs separately, as well as the ratio of the total number of commits that these pairs co-changed over the total number of commits of one of these five files. Table 4 and 5 report these statistics for the two projects. In Apache Derby, we can observe that it is easier for a model to predict co-changed pairs (93% recall on average), but the model also produces false positives that degrades the performance in terms of precision (62% on average). On the contrary, in Apache CXF, both precision and recall values are below 50%.

5. Discussion

The findings on two Apache projects show that 1) the performance evaluation strategy greatly affect the findings on co-change prediction, 2) although we use the same model construction and performance evaluation methods, it is possible to achieve contradictory findings on different projects and file sets, and 3) the performance of a co-change prediction model seem to be not related to the amount of prior changes on file pairs, but might related to the characteristics/features of co-changed files. In this section, we discuss each of these observations below.

Table 3. Performance evaluation of the predictions

Project	File ID	Precision (%)	Recall (%)
Apache Derby	1	63,43	75,89
	2	57,14	82,05
	3	25,00	100,00
	4	55,42	55,42
	5	45,00	69,23
	Avg.	49,20	76,52
	Avg (as in [3])	92,83	89,00
Apache CXF	1	33,12	53,24
	2	42,34	66,11
	3	56,00	53,08
	4	10,00	50,00
	5	62,33	71,00
	Avg.	40,75	58,69
	Avg (as in [3])	89,79	88,42

As mentioned in Section 4.5, we used two different approaches for calculating the precision and recall values of the predictors. Table 3 reports the statistics at commit level, e.g., for each commit in which file #1 was changed (SQLState.java in Derby), the number of correct predictions for its pairs (messages.xml and messages_en.properties) are aggregated to calculate the precision. Based on the statistics in Table 4 for the file #1, 248 commits were made, 180 of which were with messages.xml whereas 52 of which were with messages_en.properties files. Table 3 reports the precision of file #1 over these 248 commits as 63%. This indicates 63% of predictions correctly highlighted whether each of these two files were co-changed with file #1. On the other hand, using Wiese et al. [3]'s approach, we could observe that the models used for predicting two file pairs associated with file #1 (SQLState.java) are very successful on their own. Over 180 commits, 97% of predictions correctly highlighted that the first pair co-changed together.

Overall we could say we were able to confirm the findings in [3]. On the other hand, we observe that these rates are biased towards the granularity, i.e., commit level or file-pair level. This difference points out that it is more challenging to predict all entities that will be co-changed in a single commit than to predict whether a particular entity will be co-changed with the target file in a single commit. According to our point of view, it is more useful to succeed the former than the latter. More analysis and different modelling approaches could be needed to make multiple predictions during a commit.

Nevertheless, we do not claim that one modelling and analysis approach would yield the best results in all circumstances. For instance, in Apache Derby, both Wiese et al. [3]'s and our findings at file-pair level (Tables 4 and 5) reach very high precision and recall values. However, the same approach performed significantly worse in CXF. For almost all file pairs of CXF, the predictors using commit and developer communication features could not succeed even as good as a random classifier. The precision values are relatively better than recall values in CXF. But in particular, file #3 (JAXRSClientServerBookTest.java) and file #4 (HTTPConduit.java) got 21% and 10% precision indicating high false positives. We first argue that such differences could be due to training sample size, because in these aforementioned pairs, only 80% of 22 and 33 instances existed in the training sets of the associated models respectively, and random forest classifier might need a bigger sample to make more accurate predictions. On the other hand, although a similar scenario exists in file #5 (WSS4JInInterceptor.java) and its pair (UsernameTokenInterceptor.java), i.e., the model trained on this pair consists of around 30 instances, it has achieved 100% precision. The reasons of high false positives, and hence, low precision rates need to be further investigated. On the other hand, the reasons of low recall rates in CXF project might be explained by analysing the characteristics of co-changed entities.

Table 4. Performance measures on the selected co-changed file pairs of Apache Derby

File name	Co-changed file name	Precision	Recall	file pairs co-changed / total no. commits on the selected file
…/SQLState.java	…/messages.xml	97,30	97,30	180 / 248
	…/messages_en.properties	90,91	90,91	52 / 248
	Average	94,10	94,10	
…/DataDictionaryImpl.java	…/EmptyDictionary.java	76,47	76,47	41 / 237
	…/DD_Version.java	66,67	94,12	56 / 237
	…/SystemProcedures.java	69,23	75,00	38 / 237
	…/DataDictionary.java	78,72	90,24	90 / 237
	…/sqlgrammar.jj	55,55	100,0	39 / 237
	Average	69,13	87,17	
…/sqlgrammar.jj	…/QueryTreeNode.java	50,00	80,00	35 / 223
	…/SelectNode.java	54,54	100,0	36 / 223
	…/DataDictionaryImpl.java	61,54	80,00	39 / 223
	Average	55,36	86,67	
…/_Suite.java	…/derby_lang.runall	25,00	100,0	46 / 201
	Average	25,00	100,0	
…/DRDAConnThread.java	…/DDMReader.java	66,67	100,0	26 / 172
	…/Database.java	68,75	91,67	26 / 172
	…/NetworkServerControlImpl.java	64,71	100,0	31 / 172
	…/DDMWriter.java	66,67	100,0	29 / 172
	…/DRDAStatement.java	80,95	85,00	37 / 172
	…/AppRequester.java	38,46	100,0	26 / 172
	Average	64,37	96,11	
	Average of the repository	61,59	92,81	

For example, file #1 (JAXRSUtils.java) has in total seven files in its co-changed file set. Although the models trained on each of these seven pairs are able to achieve reasonably high precision rates, they could only be able to catch 24% of actual co-change cases in the associated test sets. This might indicate that the features used to predict co-changed entities in some projects and file pairs are not explanatory. Although Wiese et al. [3] highlighted commit and developer communication network contexts as good indicators, it could be possible that each project and file pair depicts a different dependency, and such dependencies could only be captured with different contextual information from issue and code repositories.

6. Threats to Validity

Replications help addressing both internal and external validity issues [14]. We replicated the original study [3] in an external setting, and observed that using the same projects but following with a different experimental protocol; findings could significantly vary in the context of co-changed files prediction. In terms of internal validity, we observe that the findings in both the original study and our replication hold only under specified conditions. The set of features, model construction methodology and evaluation criteria used in an empirical study could lead to inconsistent findings. For instance, we picked the top five files to build separate predictors on those with their co-changed file set. We specifically chose the most frequently edited files throughout the development history so that the models could have reasonable amount of data for training. This selection might jeopardize the findings achieved, specifically in cases of very high precision and recall values. However, if we built models for all files, we might have ended up with having files, which do not have any other files being co-changed. Therefore, in order to build a co-change predictor, we had to choose the most active files in the software projects.

Table 5. Performance measures on the selected co-changed file pairs of Apache CXF

File name	Co-changed file name	Precision	Recall	file pairs co-changed / total no. commits on the selected file
…/JAXRSUtils.java	.../JAXRSUtilsTest.java	100,0	29,87	60 / 228
	…/JAXRSInvoker.java	67,86	23,46	41 / 228
	…/ProviderFactory.java	89,19	23,24	54 / 228
	…/JAXRSOutInterceptor.java	94,87	32,46	49 / 228
	…/InjectionUtils.java	100,0	24,84	58 / 228
	…/JAXRSInInterceptor.java	93,10	21,77	49 / 228
	…/AbstractJAXBProvid41.java	91,30	14,58	35 / 228
	Average	90,90	24,32	
…/AbstractBindingBuilder.java	…/SymmetricBindingHandler.java	70,27	34,21	76 / 208
	…/AsymmetricBindingHandler.java	43,75	42,42	69 / 208
	…/TransportBindingHandler.java	75,75	34,72	68 / 208
	Average	63,26	37,12	
…/JAXRSClientServerBookTest.java	…/BookStore.java	27,27	75,00	101 / 200
	…/JAXRSUtils.java	14,28	55,88	33 / 200
	Average	20,77	65,44	
…/HTTPConduit.java	…/HTTPConduitURLEasyMockTest.java	10,00	50,00	22 / 181
	Average	10,00	50,00	
…/WSS4JInInterceptor.java	…/AbstractBindingBuilder.java	50,00	6,67	35 / 177
	…/UsernameTokenInterceptor.java	100.0	11,65	30 / 177
	Average	75,00	9,16	
	Average of the Repository	46,80	36,60	

To filter the files co-changed with the top-5 files, we took into consideration confidence values. A confidence ratio of 0,15 indicates that the training set for a model (a,b) consists of 100 changes on file *a*, and 15 of those 100 changes also includes file *b*. We referred to defect prediction models while choosing this rate, as in defect prediction the ratio of defective and defect-free instances is around 15-25% in public datasets and a machine learning classifier could roughly detect 70% of the defective instances.

The algorithm we chose in this study is random forest, which is the same as in [3]. We intentionally chose the same classifier to eliminate the effect of multiple factors over our findings. We also conducted analyses with support vector machine and logistic regression and checked if they performed better. In Apache Derby, the best performance achieved with these additional classifiers was 43% and 55% in terms of precision and recall. Therefore, we continued with the random forest classifier in this replication study.

7. Conclusion

In this study, we performed a conceptual replication of the original study by Wiese et al. [3] by modifying the experimental protocol regarding the list of features, training data generation and evaluation criteria. The results confirm that we could achieve around 90% precision and recall rates using commit and developer communication information on Apache Derby and CXF projects. However, these ratios are highly dependent on the level of granularity in which performance metrics are calculated. If a developer wants to see during a commit, which files will also be changed with the file that she is currently editing, predictor models could only recommend 59-76% of the files.

We also observe that these findings might be dependent on the project characteristics, file characteristics and/or other aspects. In order to improve models that predict co-changes, we suggest looking at other contextual information at project level and at file level, e.g. distinguishing core files that are changed with many files in the system from the files that are often co-changed with a subset of files. Although the closeness and betweenness measures were used to depict developer relations, it could also be useful to construct such dependency at file level to incorporate those into the prediction models.

8. References

[1] Borg, M., Wnuk, K., Regnell, B., Runeson, P. Supporting change impact analysis using a recommendation system: an industrial case study in a safety- critical context. *IEEE Transaction on Software Engineering*, 2017, 43(3), 675-700.

[2] Kagdi, H., Maletic, J.I. Combining Single-version and Evolutionary Dependencies for Software-change Prediction, fourth International Workshop on Mining Software Repositories (MSR'07), Minneapolis, MN, USA, 2007, pp 17.

[3] Wiese, I.S., Ré, R., Steinmacher, I., Kuroda, R.T, Oliva, G.A., Treude C., Gerosa, M.A. Using contextual information to predict co-changes. *Journal of Systems and Software*, 2017, 128(C), 220-235.

[4] Ball, T., Kim, J., Porter, A.A., Siy, H.P. If Your Version Control System Could Talk, ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering, 1997.

[5] Gall, H., Hajek K., Jazayeri, M. Detection of Logical Coupling Based on Product Release History, proceedings of the International Conference on Software Maintenance (ICSM '98), IEEE Computer Society, Washington, DC, USA, 1998, pp 190-.

[6] Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A. Mining Version Histories to Guide Software Changes, proceedings of the 26th International Conference on Software Engineering (ICSE '04), IEEE Computer Society, Washington, DC, USA, 2004, pp 563-572.

[7] Canfora, G., Cerulo, L., Cimitile, M., Penta, M.D. How changes affect software entropy: an empirical study. *Empirical Software Engineering*, 2014, 19(1), 1-38.

[8] Hassan, A.E., Holt, R.C. Predicting Change Propagation in Software Systems, proceedings of the 20th. IEEE International Conference on Software Maintenance, Chicago, IL, USA, 2004, pp 284-293.

[9] Macho, C., McIntosh, S., Pinzger, M. IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, Suita, Japan, 2016, pp 541-551.

[10] Kouroshfa, E. Studying the Effect of Co-change Dispersion on Software Quality, proceedings of the 2013 International Conference on Software Engineering (ICSE '13), Piscataway, NJ, USA, 2013, pp 1450-1452.

[11] Meyer, B., Nordio M. 2012. Empirical Software Engineering and Verification: International Summmer Schools; Springer-Verlag: Berlin, Heidelberg, 2012.

[12] Shepperd, M., Ajienka, N., Counsell, S. The role and value of replication in empirical software engineering results. *Information and Software Technology,* 2018, 99, 120-132.

[13] Silva, F.Q., Suassuna, M., França, A.C., Grubb, A.M., Gouveia, T.B., Monteiro, C.V., Santos, I.E. Replication of empirical studies in software engineering research: a systematic mapping study. *Empirical Software Engineering*. 2014, 19(3), 501-557.

[14] Shull, F., Carver, J., Vegas, S., Juristo, N. The role of replications in empirical software engineering. *Empirical Software Engineering*, 2008, 13, 211-218.