Predicting Build Co-Changes with Source Code Change and Commit Categories

Christian Macho
Software Engineering Research Group
University of Klagenfurt
Klagenfurt, Austria
Email: christian.macho@aau.at

Shane McIntosh
Dept. of Electrical and Computer Engineering
McGill University
Montreal, Canada
Email: shane.mcintosh@mcgill.ca

Martin Pinzger
Software Engineering Research Group
University of Klagenfurt
Klagenfurt, Austria
Email: martin.pinzger@aau.at

***Abstract*—When software is maintained and evolved the build configuration also needs to be updated. Knowing when to update the build configuration is typically done manually with the risk of missing an update and breaking the build. To mitigate this risk, previous work has investigated prediction models to help developers to identify commits that will likely involve an update of the build configuration.**

In this paper, we investigate whether we can improve these existing prediction models by taking into account detailed information on source code changes and commit categories. Our main hypothesis is that such detailed information on changes will significantly improve the prediction of build co-changes.

To that extent, we extract information on changes from 10 Java open source projects and use a random forest classifier to train models that predict build co-changes within and across projects. Our results show significant improvements over existing prediction models: the AUC for intra- and cross-project prediction improves by 11.54% and 9.46% respectively. In addition, we investigate advanced resampling techniques to explore the effect of unbalanced data on our models. The results show that SMOTE can particularly improve prediction models with low performance that were trained on unbalanced data. Our models improve the prediction and enable a better understanding of build co-changes.

I. Introduction

As today's software engineering projects get more and more complex, also the way how to build a project increases in complexity. Similar to the maintenance of source and test code, the build configuration needs to be maintained [1] to avoid breaking the build and slowing down the development process. Indeed, Seo *et al.* [2] identify neglected build maintenance as the most common root cause for build breakage. Kerzazi *et al.* found that such build breakages can be expensive [3].

To that extent, it is crucial to understand when changes to a build configuration are needed, meaning when changes in the source and test files also involve a change in the build files. Addressing this issue, McIntosh *et al.* [4] propose to use machine learning to train models to predict build changes based on prior build changes and changes in the source and test files. This approach has been further investigated by Xia *et al.* [5] to assess the possibility of performing such predictions across projects.

In this study, we aim to enhance their model by using source code change and commit categories. We use Change Distiller [6], [7] to obtain the source code change categories. Change Distiller parses two consecutive versions of a Java file and creates their ASTs. The two ASTs are compared using a tree-differencing algorithm [8]. The differences in the ASTs are then mapped to their corresponding source code change categories. For the categorization of commit messages, we follow the approach of Hattori *et al.* [9], who divide commits into four categories based on the content of the commit message.

Using the source code change and commit categories, and the basic attributes of [4] computed for ten Java open source projects, we train intra- and cross-project prediction models with the random forest classifier. For intra-project prediction, the data of a single project is split into training and test sets, whereas for cross-project prediction, one project is used for training the classifier and another project is used for testing the model. We also investigate various resampling methods to deal with the issue of unbalanced data for training prediction models. Finally, we report on two examples of source code changes that involved changes in build files to better understand the potential impact of code changes on the build configuration.

With the results of our study, we aim to answer the following three research questions:

(RQ1) To what extent can source code change and commit categories improve intra- and cross-project build co-change prediction?
Source code change and commit categories significantly improve the model for predicting build co-changes. Compared to the models presented by McIntosh *et al.* [4], the average AUC value for intra-project prediction improves from 0.78 to 0.87 or 11.54%. Regarding cross-project prediction, compared to the models presented by Xia *et al.* [5] our results show an average improvement of the AUC values from 0.74 to 0.81 or 9.46%.

(RQ2) To what extent can advanced resampling methods improve the performance of build co-change prediction?
Advanced resampling methods (*e.g.*, SMOTE [10]) improve the performance of intra- and cross-prediction models. In particular, the prediction models with the weakest performance among the studied Java projects

yield the largest improvement. For instance, the AUC value for the Hadoop project improved from 0.73 to 0.89 for intra-project prediction. For cross-project models that are trained using Hadoop data, the AUC value when testing using the Karaf project improved from 0.50 to 0.88.

(RQ3) Which attributes of our models are important to predict build co-changes within and across projects? *Number of Files*, *Method Body Changes*, and *Prior Build Co-Changes* are the most important attributes for building intra- and cross-project prediction models, directly followed by the commit categories *Management*, *Forward Engineering*, and *Reengineering*.

By answering these research questions, we make the following main contributions: (1) An improved model for build co-change prediction with source code change and commit categories; (2) an evaluation of the models for intra- and cross-project prediction; (3) an evaluation of advanced resampling methods for addressing the issue of unbalanced data in the training set; and (4) a qualitative study to understand the impact of source code changes on build co-changes.

The remainder of the paper is organized as follows: Section II sums up the related work about build systems and prediction. Section III describes our data extraction process. Sections IV, V, and VI present the results of our analyses, which address our three RQs. In Section VII we discuss the threats to validity and we draw conclusions in Section VIII.

II. Related Work

In this section, we discuss prior studies on build systems and change prediction.

Build Systems. Several prior studies have focused on analyzing build maintenance. For example, Kumfert *et al.* [11] and Hochstein *et al.* [12] show that build maintenance generates a "hidden overhead" on software development. Furthermore, Adams *et al.* [1] claim that source and build code tend to co-evolve and show this tendency in a study with the Linux kernel. Furthermore, McIntosh *et al.* confirmed the observations of Adams *et al.* in Java build systems [13], ANT build systems [14], and also for other languages [15]. Xia *et al.* [16] and Zhou *et al.* [17] focus on detecting and automatically inferring missing dependencies. Further studies aim at understanding build changes and predicting build co-changes, such as McIntosh *et al.* [4] within a project and Xia *et al.* [5] across projects.

There are also tools that support build maintenance. Adams *et al.* [18] propose a reverse engineering tool called MAKAO based on building a dependency graph. Tamrawi *et al.* [19] present SYMake, which aims to visualize dependencies in Makefiles. Furthermore, MkDiff by Al-Kofahi *et al.* [20] extracts the semantics of build specification changes. Hardt *et al.* [21], [22] present a tool called Formiga which is used to automate build changes or to assist in build refactoring of ANT build files.

Change Prediction. A large body of research exists that uses prediction methods, in particular machine learning, to understand various characteristics of software engineering. For instance, Hassan *et al.* [23] predict the passing of a certification process by using decision trees. Ratzinger *et al.* [24] use classification algorithms to predict the need of future refactoring. Ibrahim *et al.* [25] predict whether a developer should contribute to an email discussion. Knab *et al.* [26] predict defect densities in source code files. Furthermore, Romano *et al.* [27] use classifiers to identify change-prone Java interfaces. Finally, Giger *et al.* [28] use fine-grained source code changes for predicting future bugs.

We motivate our research by acknowledging that source code changes have already been used for various prediction models and the fact that source code and build code co-evolve. We claim that source code changes are indicators for build co-changes and investigate this in this paper.

III. Predicting Build Co-Changes

In this section, we present our approach to predict build co-changes. The approach is split up into two main parts: Data Extraction and Data Analysis. The data extraction part deals with the extraction of work items from source code repositories and the calculation of various measures used for building the prediction models. The data analysis part builds the prediction models and measures their performance. Figure 1 shows an overview of our approach. Below we describe each part in detail.

A. Data Extraction

At the beginning of our data extraction, we clone a source code repository and retrieve the log information on each commit from the master branch. We currently support software projects that use git[1] as source code repository and Maven[2] as build system (see Section III-C). Next, we iterate over each commit and perform the following steps:

File Type Classification. In our approach, we distinguish between build, source, and test files. As Maven provides a strong convention for the repository structure, we directly rely on this project layout. Build files are all files named `pom.xml`. Source and test files are distinguished by investigating their respective location in the project folders. Files located in the folder `src/main` are considered source files. Files located in the folder `src/test` are considered test files. We do not handle other files as our proposed model does not make use of metrics calculated with other files.

Change Extraction. One of the most important parts of the process is the change extraction. This step extends the approach of McIntosh *et al.* [4]. We compare each commit with the preceding (parent) commit and extract the fine-grained source code changes of the Java files. We use Change Distiller [7], [8] for extracting those changes. Change Distiller parses two consecutive versions of a Java file and creates an AST for each of them. Then, it compares the two ASTs and

[1]https://git-scm.com
[2]https://maven.apache.org

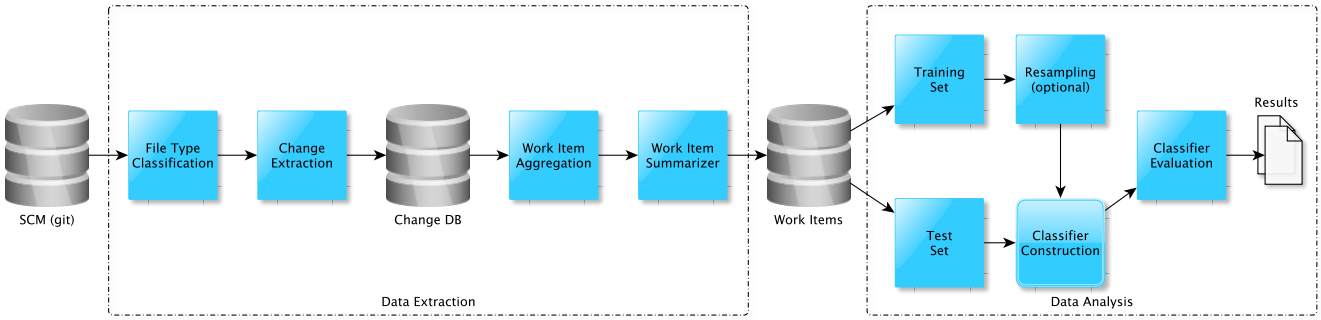


Fig. 1. Overview of our approach for predicting build co-changes

extracts the differences using a tree-differencing algorithm. Change Distiller is able to extract source code changes down to the AST level, such as the insertion of a method invocation, the change of a method return type, or the change of an if condition. Changes to the build files are extracted by checking whether a commit references `pom.xml` files that are added, deleted, or modified. The source code, test code, and build file changes are stored into a database that we call Change DB together with commit meta data (compared revisions, time, author, and commit message).

Work Item Aggregation. Because earlier studies, such as McIntosh *et al.* [29], have shown that commits are too fine-grained to properly represent a development task, we group commits that are logically coupled into work items. We use a similar approach for identifying work items as proposed by McIntosh *et al.* [29] by parsing the commit messages and searching for patterns[3] that identify a work item in the issue tracking system. We group commits by linking work items with their respective commits in our database.

Work Item Summarizer. This step calculates the various attributes for each work item as shown in Table I. Aiming at improving existing prediction models, we calculate the basic attributes proposed by McIntosh *et al.* [4] and add our measures of commit categories and source code changes. Regarding the commit categories, we implement the approach by Hattori *et al.* [9] who use keywords to divide commits into four categories: Forward Engineering, Reengineering, Corrective Engineering, and Management. For each work item, we investigate the commit messages and count the matched keywords for each category.

For source code changes, we use a similar approach. Fluri *et al.* provide a taxonomy of fine-grained source code changes and categories [6]. We adopt this taxonomy to cover all change types that Change Distiller is capable of extracting, resulting in ten source code change categories. Then, for each work item, we count the number of changes in each category. Compared to the approach by McIntosh *et al.* [4], these attributes are computed on fine-grained source changes explaining the attribute '(Source/Test File modified)' in more detail. We argue

[3]E.g.: HADOOP-[number] or HBASE-[number]; where [number] stands for a sequential number given by the issue tracking system

TABLE I
WORK ITEM ATTRIBUTES USED TO BUILD PREDICTION MODELS

Attribute Category	Name	Abbrev.
Source Code Change Categories [6]	Class Body Changes	CBC
	Method Body Changes	MBC
	Structure Statements	SST
	Access Modifier Changes	AMC
	Attribute Declaration Changes	ADC
	Class Declaration Changes	CDC
	Final Modifier Changes	FMC
	Method Declaration Changes	MDC
	Documentation Changes	DOC
	Unclassified Changes	UNC
Basic Model Attributes [4]	Number of Files	NF
	Prior Build Co-Changes	PBC
	(Source/Test) File added	(S/T)FA
	(Source/Test) File deleted	(S/T)FD
	(Source/Test) File renamed	(S/T)FR
Commit Categories [9]	Corrective Engineering	CE
	Forward Engineering	FE
	Management	MA
	Reengineering	RE

that this more detailed information leads to better prediction models. We also measure the other attributes of the basic model that was proposed by McIntosh *et al.* [4], which are '(Source/Test) File added, deleted, and renamed', as well as 'Number of Files' and 'Prior Build Co-Changes'. 'Number of Files' represents the total number of added/changed/deleted files in a work item. For 'Prior Build Co-Changes', we select the maximum ratio of all files in the work item that were build co-changing in previous work items.

Finally, we label a work item as build co-changing if at least one `pom.xml` in that work item has been added, deleted, or changed.

B. Data Analysis

The data analysis part splits the data into a training and a test set. Then, the optional step of applying resampling methods to the training set is executed. With the resulting data set we train a random forest classifier for binary classification. The classifier is evaluated on the test set based on which we compute the F-measure, AUC, and MCC to quantify the

performance of our prediction models. Below, we describe these steps in detail.

Training/Test Set separation. For the validation of our prediction models, we split the data into a training set and a test set. The split is different for intra-project prediction and for cross-project prediction. For intra-project prediction, we use repeated random sub-sampling. For each project, we repeat the experiment 100 times to achieve reliable performance metrics and to minimize the influence of selecting instances for the test set. In each run, we randomly select 90% of the work items in a project for training and the remaining 10% of work items for testing the prediction model. The values of the performance metrics are averaged over the 100 runs. This validation strategy has been used in several previous studies, such as Pinzger *et al.* [30]. For cross project prediction, we use all work items of one project as the training set and the work items of another project as test the set. This is repeated for all possible combinations of different projects.

Resampling. Similar to the systems studied by McIntosh *et al.* [4], our subject systems listed in Table II show an imbalanced number of work items that build co-changed. Since this affects the performance of prediction models, we experiment with several advanced resampling techniques to achieve an equal distribution of work items that did build co-change and that did not. Section V presents the details on the resampling techniques and their effect on the prediction performance.

Classifier Construction. Since we aim to predict whether a work item will need a build co-change or not, we construct a *binary* classifier using the random forest algorithm [31]. The random forest classifier generates many decision trees. Each decision tree is built with a random subset of all model attributes. The classifier calculates a classification decision for each of the trees and then aggregates the partial results to a total classification result. We selected random forest since this algorithm has been used in many previous empirical studies [4], [5], [32], [33], and tends to have good predictive power [34].

Classifier Evaluation. We apply the classification model to the test set and output the results in the following confusion matrix.

	Classified As	
Actual Category	Change	No Change
Change	a	b
No Change	c	d

Based on this matrix, we compute the following performance metrics:

- **Precision (P):** Ratio of work items correctly classified as build co-changing (a) out of all work items classified as build co-changing (a+c), *i.e.*, $P = \frac{a}{a+c}$.
- **Recall (R):** Ratio of work items correctly classified as build co-changing (a) out of all work items that actually did build co-change (a+b), *i.e.*, $R = \frac{a}{a+b}$.
- **F-Measure:** The harmonic mean of precision and recall, *i.e.*, $F = 2 \cdot \frac{precision \cdot recall}{precision + recall}$
- **Area under ROC-Curve (AUC):** The area under the curve plots the true positive rate ($\frac{a}{a+b}$) against the false positive rate ($\frac{c}{c+d}$) for various values of the chosen threshold used to determine whether a work item is classified as build co-changing. Values of AUC range from 0 (worst classifier performance) to 1 (best classifier performance) where 0.5 indicates that the classifier is no better than random guessing.
- **Matthews Correlation Coefficient (MCC):** Measures the quality of a binary classification and is considered a good metric for measuring the performance of classifiers [35]. It is calculated as $MCC = \frac{a \cdot d - c \cdot b}{\sqrt{(a+c) \cdot (a+b) \cdot (d+c) \cdot (d+b)}}$ and values range from -1 (total disagreement) to +1 (perfect classifier) whereas 0 means the classifier is no better than random guessing.

We use these performance metrics to compare our approach with the approaches of McIntosh *et al.* [4] and Xia *et al.* [5].

C. Studied Systems

To evaluate the proposed model and approach, we selected Java open source projects of different sizes and domains that satisfy the following constraints:

- uses **git** as source code management system;
- uses **Apache Maven** as its build system;
- satisfies the **Maven Standard Directory Layout;**[4]
- provides at least **two years of evolution data**;
- shows **similar distribution of commits and work items**

We investigated several subject systems and finally selected 10 projects for our experiments. Table II lists the selected systems and their properties that are relevant for this study. Among the systems, we selected Java frameworks, such as Hibernate Search, Karaf, Camel, and Wicket, as well as end-user systems, such as Jenkins, ActiveMQ, Wildfly, and Roo. Furthermore, we selected two projects from the Hadoop system, namely Hadoop and HBase.

Regarding the links between commits and change requests, we compared the distribution of commits and work items per month. Similar to McIntosh *et al.* [4] we used beanplots [36] to visualize the distribution of commits and work items per month. Figure 2 shows the shape of the distributions for each project. We selected projects that show similar distributions of commits and work items over time, meaning that commits are adequately linked to work items. A bad linkage would lead to a biased data set which is a known problem for building prediction models [37], [38].

IV. Improving build co-change prediction

In this section, we address the first research question: *(RQ1) To what extent can source code change and commit categories improve intra- and cross-project build co-change prediction?* By comparing our results to those of McIntosh *et al.* [4] and Xia *et al.* [5], we show that fine-grained source code changes and commit categories can lead to improved models

[4]https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

TABLE II
COMMIT PROPERTIES OF THE STUDIED PROJECTS

	ActiveMQ	Hadoop	HBase	Camel	Hibernate Search	Wicket	Wildfly	Karaf	Roo	Jenkins
First Commit	2007	2011	2011	2007	2008	2006	2010	2010	2009	2012
Project Files	11045	21027	8346	9594	7456	33377	33804	4638	5676	9126
Commits	6768	10753	8008	4929	3899	12814	16924	3929	4878	6031
Work Items	2542	9441	6209	1336	1217	3623	5628	2018	2094	970
Commits with Work Items	60%	98%	94%	60%	83%	44%	52%	83%	95%	31%
Build Co-Changing Work Items	375	543	397	266	400	130	1314	915	281	109
Not Build Co-Changing Work Items	2167	8898	5812	1070	817	3493	4314	1103	1813	861
Build Co-Change Ratio	15%	6%	6%	20%	33%	4%	23%	45%	13%	11%

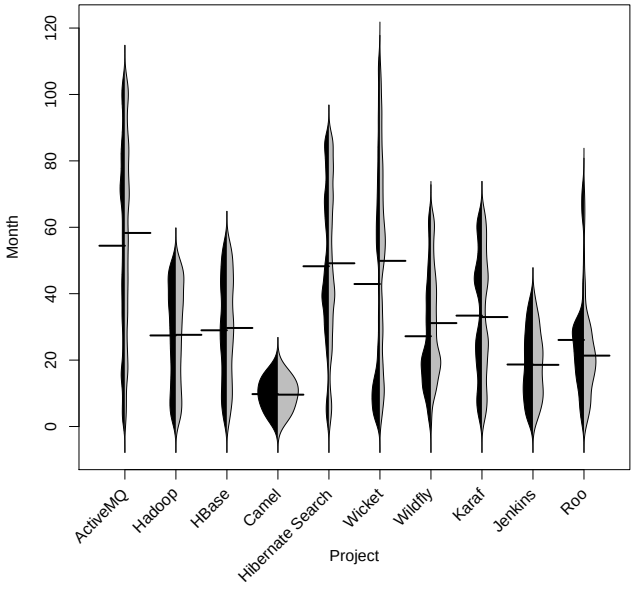


Fig. 2. Comparison of commits and work items per months (black denotes commit distribution and gray denotes work item distribution)

for predicting build co-changes within and across projects. In the following, we first present the results for the case of intra-project prediction and then for cross-project prediction.

A. Intra-Project Prediction

We apply our approach to the selected Java open source projects to extract the data, measure the metrics, and train the prediction models using the random-forest classifier. For the comparison with the approach presented by McIntosh *et al.* [4], we first compute the models with the basic model attributes that we then extend with our measures of the commit and source code change categories as listed in Table I. Concerning the validation of the prediction models, we use a repeated random sub-sampling approach as described in Section III.

Table III shows the results of the classification models computed with our approach and the approach of McIntosh *et al.* [4]. The metric values have been averaged over the 100 runs. At the bottom of the table, we also present the minimum, maximum, and average values over all projects. Figure 3 shows a detailed comparison of the results with box-plots.

TABLE III
PERFORMANCE METRICS OF PREDICTION MODELS COMPUTED WITH OUR APPROACH (O) AND THE APPROACH OF MCINTOSH *et al.* [4] (M).

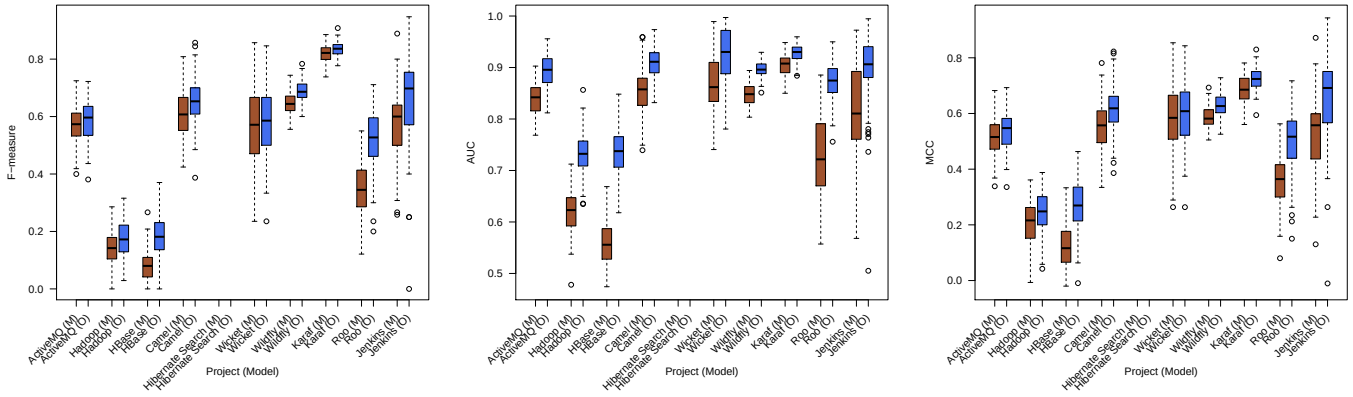
Project	F-measure			AUC			MCC		
	M	O	d	M	O	d	M	O	d
ActiveMQ	0.57	0.60	0.10	0.84	0.90^{2}	0.75	0.52	0.55^{1}	0.19
Hadoop	0.14	0.17^{2}	0.29	0.62	0.73^{2}	0.95	0.22	0.25^{2}	0.29
HBase	0.08	0.18^{2}	0.74	0.56	0.74^{2}	1.00	0.12	0.27^{2}	0.76
Camel	0.61	0.65^{2}	0.33	0.86	0.91^{2}	0.70	0.56	0.62^{2}	0.38
Hib. Search	0.65	0.72^{2}	0.62	0.77	0.89^{2}	0.95	0.52	0.63^{2}	0.67
Wicket	0.57	0.59	0.09	0.86	0.93^{2}	0.54	0.58	0.61	0.09
Wildfly	0.64	0.69^{2}	0.59	0.85	0.90^{2}	0.94	0.58	0.63^{2}	0.55
Karaf	0.82	0.84^{2}	0.32	0.91	0.93^{2}	0.62	0.69	0.72^{2}	0.44
Roo	0.34	0.53^{2}	0.79	0.72	0.87^{2}	0.95	0.36	0.52^{2}	0.71
Jenkins	0.60	0.70^{2}	0.40	0.81	0.91^{2}	0.57	0.56	0.69^{2}	0.56
MIN	0.08	0.17	0.09	0.56	0.73	0.54	0.12	0.25	0.09
MAX	0.82	0.84	0.79	0.91	0.93	1.00	0.69	0.72	0.76
AVG	0.50	0.57	0.43	0.78	0.87	0.80	0.47	0.55	0.47

$^{1}p < 0.05$; $^{2}p < 0.001$: significance level of the Two-Tailed Mann-Whitney U-Test
d: effect size computed with Cliff's Delta

Looking at the performance metrics and the box-plots, we can see that the prediction models computed with our approach outperform the models computed with the approach by McIntosh *et al.* [4]. Regarding the F-measure, we can improve the minimum value from 0.08 to 0.17 and the maximum value from 0.82 to 0.84 with an average increase from 0.50 to 0.57. The minimum AUC enhances from 0.56 to 0.73 and the maximum value from 0.91 to 0.93 resulting in an average improvement from 0.78 to 0.87. For the MCC metric, we observe similar behavior. Minimum MCC increases from 0.12 to 0.25, maximum from 0.69 to 0.72, and the average improves from 0.47 to 0.55.

To test whether the improvements obtained with our prediction models are not by chance, Table III shows the results of a Two-Tailed Mann-Whitney U-Test and the Cliff's-Delta d. Except the F-measures for the projects ActiveMQ and Wicket, and the MCC value of Wicket, the differences are statistical significant. For Cliff's Delta d, the effect size is considered negligible for $d < 0.147$, small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.47$, and large for $d \geq 0.47$ [39]. Looking at the AUC values, the effect sizes of all projects are considered large. For the F-measures the difference is considered large for 4 out of 10 projects, the MCC values show a large difference for 5 out of the 10 projects.

Based on these results, in particular the large improvement of the AUC values, we can answer the first research question



(a) F-measure (b) AUC (c) MCC

Fig. 3. Comparison of F-measure, AUC, and MCC values of prediction models computed with McIntosh *et al.*'s [4] basic (M) and our attribute set (O)

for the case of intra-project prediction:

Source code change and commit categories significantly improve the model for intra-project prediction of build co-changes resulting in a gain of the average AUC of 11.54% compared to the model of McIntosh et al. [4].

B. Cross-Project Prediction

For this experiment, we use the same data set as before but apply a different method for obtaining the training and test sets. In particular, we use each project once for training the random forest classifier. The resulting prediction model is then tested with the data of each of the other nine projects. For instance, we train a prediction model with the Hadoop work items and then test the model on the sets of work items of each other project. The values of performance metrics are averaged over all tested projects.

Table IV lists the average F-measure, AUC, and MCC values that are computed for each project when it is used to train a prediction model (Source) evaluated with the other nine projects, and when it is used for testing the prediction models that are built with each of the other nine projects (Target). Looking at the Source values, we can see that, except Hadoop, HBase, Roo, and Jenkins, each project can be used to train prediction models with an average F-measure ≥ 0.51, AUC ≥ 0.85, and MCC ≥ 0.45. As a target, only Hibernate Search and Wildfly have an average AUC ≤ 0.79. Regarding the F-measure of using projects to test prediction models (*i.e.*, targets), only the projects Hadoop and HBase have values < 0.40.

Table V lists an excerpt of the detailed cross-project prediction results. Several prediction models show high values for the performance metrics. For example, training a model with the data of Wildfly and testing it on Karaf reaches an F-measure of 0.80, an AUC of 0.92, and an MCC of 0.70. We can observe a similar performance for using Camel as training set and Karaf as test set (F-measure of 0.80, AUC of 0.91, and MCC of 0.67). In total, there are 90 combinations of projects that we investigated. Four combinations show F-measures ≥ 0.75 having three of those with F-measure ≥ 0.80. 38 combinations achieve an AUC ≥ 0.85 with 9 combinations having an AUC ≥ 0.90. Concerning MCC, 6 combinations have an MCC ≥ 0.6 having 2 projects with an MCC ≥ 0.7.

Comparing our results with the average performance values for cross-project prediction reported by Xia *et al.* [5], we note an improvement of the F-measure from 0.40 to 0.41 and a significant improvement of the AUC value from 0.74 to 0.81 (9.46%). This underlines the predictive power of commit and source code changes categories also for cross-project build

TABLE IV
PERFORMANCE METRICS OF CROSS-PROJECT PREDICTION. EACH PROJECT IS USED TO TRAIN (SOURCE) AND TEST (TARGET) PREDICTION MODELS

Project	Source			Target		
	F-measure	AUC	MCC	F-measure	AUC	MCC
ActiveMQ	0.58	0.88	0.52	0.38	0.80	0.38
Hadoop	0.16	0.64	0.22	0.31	0.81	0.30
HBase	0.20	0.76	0.27	0.34	0.81	0.33
Camel	0.56	0.87	0.50	0.44	0.80	0.45
Hibernate Search	0.56	0.86	0.51	0.46	0.75	0.39
Wicket	0.51	0.85	0.47	0.40	0.86	0.42
Wildfly	0.58	0.87	0.52	0.43	0.79	0.42
Karaf	0.52	0.86	0.45	0.52	0.82	0.47
Roo	0.24	0.70	0.25	0.43	0.84	0.39
Jenkins	0.22	0.83	0.26	0.43	0.85	0.41
MIN	0.16	0.64	0.22	0.31	0.75	0.30
MAX	0.58	0.88	0.52	0.52	0.86	0.47
AVG	0.41	0.81	0.40	0.41	0.81	0.40

TABLE V
DETAILED CROSS PREDICTION RESULTS (EXCERPT)

Source	Target	F-measure	AUC	MCC
Camel	Wildfly	0.67	0.87	0.58
Wildfly	Karaf	0.80	0.92	0.70
Hibernate Search	Wicket	0.56	0.92	0.54
Camel	Karaf	0.80	0.91	0.67
ActiveMQ	Wicket	0.55	0.94	0.53
ActiveMQ	Karaf	0.80	0.92	0.70

co-change prediction. Based on these results, we can answer RQ1 for the case of cross-project prediction:

Source code change and commit categories improve the model for cross-project prediction of build co-changes resulting in a gain of the average AUC of 9.46% when compared to the model of Xia et al. [5].

V. Applying Advanced Resampling Methods

Most classifiers, including random forest, focus on correctly classifying the majority class because this yields better classification performance. Thus, classifier performance on minority class instances tends to suffer. As shown in Table II, build co-changing work items are the minority class. Indeed, our data set is unbalanced with respect to the work items that build co-changed. In this section, we address this issue and investigate several resampling methods to answer the second research question: *(RQ2) To what extent can advanced resampling methods improve the performance of build co-change prediction?*

He *et al.* [40] provide an overview of state-of-the-art methods for advanced resampling to achieve a more balanced data set for training prediction models. This overview includes Synthetic Minority Over-sampling TEchnique (SMOTE) [10], Edited Nearest Neighbors (ENN) [41], and TomekLinks [42]. SMOTE creates artificial instances with respect to the model attributes. ENN and TomekLinks remove instances that are prone to diminish the classifier performance. Batista *et al.* [43] investigated the performance of those methods and combinations of them. Based on their results, they recommend combinations of SMOTE with either ENN or TomekLinks.

Below, we report on the results that we obtain by repeating previous experiments on intra- and cross-project prediction with the various resampling methods. In the remainder of the paper, we refer to the methods as NO=no resampling, S=SMOTE, T=TomekLinks, E=ENN, ST=SMOTE and TomekLinks, and SE=SMOTE and ENN. We apply the resampling only to the training data set and compare the performance of resulting prediction models using the F-measure, AUC, and MCC.

A. Intra-Project Prediction

Table VI shows the results when applying resampling for training intra-project prediction models to our selected projects. Similar to Section IV we use repeated random subsampling with 100 runs to validate the prediction models. Values have been averaged over the 100 runs.

Comparing the values, we observe the highest improvements for Hadoop and HBase. Note, the prediction models that we computed for these two projects showed the lowest performance in our experiments before. The F-measure for these projects increases from 0.17 to 0.43 (Hadoop) and from 0.18 to 0.40 (HBase). Their AUC increases from 0.73 to 0.89 (Hadoop) and from 0.74 to 0.87 (HBase). Similarly, their MCC increases from 0.25 to 0.39 (Hadoop) and from 0.27 to 0.37 (HBase). For the other projects, the resampling methods lead to no or only a small improvement. For some projects, the values show even a small decrease in the performance, such as the AUC values for the ActiveMQ project.

Regarding the resampling methods, the SMOTE (S) alone or in combination with either TomekLinks (ST) or ENN (SE) show the best performance. This supports the recommendation of Batista *et al.* [43] to use SMOTE combined with ENN or TomekLinks. In particular for the Hadoop and HBase projects, they lead to a considerable improvement of the F-measure, AUC, and MCC. For instance, using SMOTE (S), the F-measure of Hadoop increases from 0.17 to 0.43, the AUC from 0.73 to 0.89, and the MCC from 0.25 to 0.39. Using TomekLinks (T) or ENN (E), only small improvements (if at all) are observed.

Comparing the averages, we see the highest improvement is achieved through SMOTE (S) and SMOTE in combination with TomekLinks (ST) or ENN (SE). For S, ST, and SE, the average F-measure increases from 0.57 to 0.61, for S and SE the average AUC increases from 0.87 to 0.90, and to 0.89 for ST. Regarding the average MCC, the resampling methods do not show an improvement. Based on these results, we answer RQ2 for the case of intra-project prediction:

Advanced resampling methods improve the classifier performance of intra-project prediction. The biggest improvement is obtained with SMOTE or combinations of it and for models with low performance.

B. Cross-Project Prediction

We apply the advanced resampling techniques to cross-project prediction. We train the classifier with the resampled data of each project as the training set and use each of the other projects as the test set. Note, the test set is never resampled. We then compare the performance of the resulting prediction models using the F-measure, AUC, and MCC. As we retrieve a large amount of data from this experiment, we only present the results when using Hadoop as source project to train the prediction models in Table VII. These models show the biggest improvement in the performance obtained through resampling. The results of the other projects are available online.[5]

We can see that SMOTE (S) and combinations of SMOTE with ENN (SE) and TomekLinks (ST) improve classification performance. For instance, when predicting Karaf, the F-measure increases from 0.11 to 0.71 (ST), the AUC from 0.50 to 0.88 (S), and the MCC from 0.17 to 0.59 (ST). The cross-project prediction from Hadoop to Wildfly show similar improvement - the F-measure increases from 0.16 to 0.61 (ST, SE), the AUC value from 0.57 to 0.84 (S, ST, SE), and the MCC from 0.23 to 0.55 (ST, SE). Using TomekLinks or ENN alone does not lead to such large improvements in the case of Hadoop. The average values of Table VII show similar results. The average F-measure can be raised from 0.17 to 0.52 (S,ST,SE), the average AUC from 0.65 to 0.84 (S,ST,SE), and the average MCC from 0.23 to 0.46 (S,ST,SE).

[5]http://serg.aau.at/bin/view/ChristianMacho/DataAndTools

TABLE VI
F-MEASURE, AUC, AND MCC FOR INTRA-PROJECT PREDICTION PER RESAMPLING METHOD AND PROJECT

	F-measure						AUC						MCC					
Project	NO	S	T	E	ST	SE	NO	S	T	E	ST	SE	NO	S	T	E	ST	SE
ActiveMQ	0.60	0.59	0.60	0.61	0.60	0.61	0.90	0.88	0.89	0.89	0.88	0.88	0.55	0.53	0.54	0.56	0.52	0.55
Hadoop	0.17	0.43	0.23	0.19	0.43	0.43	0.73	0.89	0.75	0.78	0.88	0.88	0.25	0.39	0.28	0.27	0.39	0.39
HBase	0.18	0.40	0.20	0.17	0.40	0.39	0.74	0.87	0.74	0.73	0.87	0.86	0.27	0.37	0.28	0.27	0.36	0.36
Camel	0.65	0.65	0.69	0.67	0.64	0.67	0.91	0.90	0.91	0.90	0.89	0.91	0.62	0.59	0.63	0.61	0.55	0.59
Hibernate Search	0.72	0.72	0.72	0.71	0.72	0.70	0.89	0.89	0.88	0.89	0.88	0.87	0.63	0.60	0.60	0.60	0.60	0.57
Wicket	0.59	0.55	0.55	0.60	0.52	0.53	0.93	0.95	0.93	0.94	0.93	0.95	0.61	0.54	0.56	0.61	0.52	0.52
Wildfly	0.69	0.69	0.70	0.70	0.70	0.70	0.90	0.90	0.90	0.90	0.89	0.89	0.63	0.62	0.64	0.64	0.61	0.61
Karaf	0.84	0.84	0.84	0.84	0.84	0.84	0.93	0.93	0.93	0.93	0.93	0.93	0.72	0.73	0.72	0.72	0.71	0.72
Roo	0.53	0.52	0.53	0.52	0.53	0.52	0.87	0.86	0.87	0.87	0.87	0.87	0.52	0.46	0.50	0.50	0.47	0.46
Jenkins	0.70	0.67	0.69	0.67	0.67	0.67	0.91	0.91	0.91	0.91	0.92	0.91	0.69	0.65	0.67	0.66	0.65	0.63
AVG	0.57	0.61	0.58	0.57	0.61	0.61	0.87	0.90	0.87	0.87	0.89	0.90	0.55	0.55	0.54	0.54	0.54	0.54

TABLE VII
F-MEASURE, AUC, AND MCC FOR CROSS-PROJECT PREDICTION PER RESAMPLING METHOD FOR THE HADOOP PROJECT

	F-measure						AUC						MCC					
Target Project	NO	S	T	E	ST	SE	NO	S	T	E	ST	SE	NO	S	T	E	ST	SE
ActiveMQ	0.13	0.44	0.16	0.13	0.44	0.44	0.66	0.83	0.67	0.66	0.83	0.83	0.23	0.40	0.25	0.23	0.39	0.39
HBase	0.20	0.40	0.25	0.22	0.39	0.40	0.66	0.84	0.66	0.66	0.85	0.85	0.27	0.36	0.31	0.29	0.35	0.36
Camel	0.24	0.51	0.28	0.25	0.55	0.53	0.65	0.82	0.65	0.65	0.83	0.82	0.31	0.51	0.35	0.32	0.53	0.51
Hibernate Search	0.14	0.61	0.19	0.19	0.61	0.62	0.52	0.78	0.52	0.52	0.78	0.78	0.17	0.50	0.19	0.21	0.50	0.50
Wicket	0.28	0.53	0.33	0.29	0.50	0.50	0.71	0.89	0.70	0.71	0.89	0.88	0.35	0.52	0.36	0.36	0.48	0.48
Wildfly	0.16	0.60	0.20	0.18	0.61	0.61	0.57	0.84	0.57	0.57	0.84	0.84	0.23	0.54	0.25	0.25	0.55	0.55
Karaf	0.11	0.70	0.15	0.13	0.71	0.70	0.50	0.88	0.50	0.50	0.87	0.87	0.17	0.58	0.20	0.18	0.59	0.58
Roo	0.28	0.41	0.30	0.30	0.42	0.42	0.83	0.83	0.83	0.82	0.83	0.83	0.35	0.35	0.36	0.37	0.34	0.35
Jenkins	0.02	0.48	0.09	0.02	0.47	0.47	0.71	0.85	0.72	0.72	0.86	0.85	-0.01	0.42	0.15	0.03	0.40	0.41
AVG	0.17	0.52	0.22	0.19	0.52	0.52	0.65	0.84	0.65	0.65	0.84	0.84	0.23	0.46	0.27	0.25	0.46	0.46

We can answer the second research question RQ2 for the case of cross-project prediction:

SMOTE and combinations of SMOTE with TomekLinks and ENN can improve cross-project prediction. Projects with low original performance benefit the most.

VI. CHARACTERISTICS OF THE PREDICTION MODEL

In this section, we set out to answer our third research question: *(RQ3) Which attributes of our models are important to predict build co-changes within and across projects?*

To answer this question, we first analyze the importance of each attribute in our intra- and cross-prediction models. We measure the importance using the mean decrease in the Gini coefficient, a measure of how each variable contributes to the homogeneity of the nodes and leaves in the resulting random forest [31]. Furthermore, we provide anecdotal evidence obtained through a manual analysis of selected changes in the source code that involved changes in the build files.

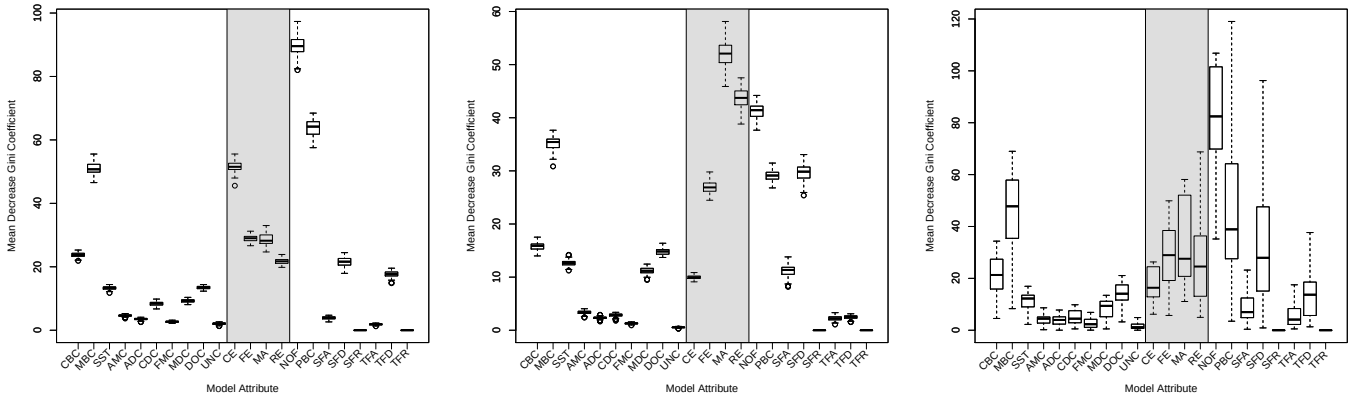
A. Importance of Attributes

First, we deal with the variable importance of the model attributes that we obtain from the random forest classifier. Figure 4 shows the box-plots of the Mean Decrease in the Gini coefficient (MDG) from the 100 runs of the intra-project prediction experiment for the two projects ActiveMQ (a) and Roo (b), and over all the ten projects (c).

According to the box-plots depicted in Figure 4a, the two attributes *Number of Files* and the *Prior Build Co-Changes* are most important for predicting build co-changes in the ActiveMQ project. They are followed by *Method Body Changes* and *Corrective Engineering* commit category. Regarding the values output for the Roo project depicted by Figure 4b, we spot the *Management* and *Reengineering* commit categories as most important attributes. They are closely followed by *Number of Files* and *Method Body Changes* attributes. Furthermore, the attributes *Source File Deleted*, *Prior Build Co-Changes*, and *Forward Engineering* are of value in these models to predict build co-changes. These results indicate that attributes from all three categories are important for building co-change prediction models.

To analyze the overall importance of the attributes, we aggregate the values of the MDG over all ten of the studied projects as shown in Figure 4c. Based on these box-plots, we observe that *Number of Files* is the most important attribute followed by *Method Body Changes* and *Prior Build Co-Changes*. Furthermore, the three commit categories *Management*, *Forward Engineering*, and *Reengineering*, as well as the *Source File Deleted* attribute are of value for predicting build co-changes. In contrast, attributes denoting changes in the access and final modifiers, attribute and class declaration, unclassified source code changes, as well as renaming of source and test files do not significantly impact the prediction.

The analysis of the models from the cross-prediction experiments obtains similar results. This is expected because the training of the random forest classifier is done with almost the same training set - instead of using 90% of the work items for training the classifier, it is trained with all work items of a project. In summary, our results confirm the findings of



(a) ActiveMQ (b) Roo (c) Overall

Fig. 4. Mean decrease Gini coefficient for attributes (see Table I) of intra-project prediction models for ActiveMQ (a), Roo (b), and over all ten projects (c)

McIntosh *et al.* [4] (*i.e.*, *Number of Files* and *Prior Build Co-Changes* are important) but also add that *Method Body Changes* and commit categories help to significantly improve the prediction models.

B. Build Co-Change Examples

In addition to the quantitative analysis of the prediction models, we manually analyzed a number of work items. We focused on work items that exhibit *Method Body Changes* and analyzed the changes in the source and build files by browsing them with the SourceTree tool.[6] Below, we report on two representative examples, the first adding a dependency to a third party Java library and the second one removing an unused dependency.

Concerning the first example, we analyzed the commit `3a6d67e8f95320bea91b7c7106173c9b34773bc5`[7] of the ActiveMQ project containing the following change in the `ActiveMQServiceFactory.java` file.

```
ResourceXmlApplicationContext ctx =
new ResourceXmlApplicationContext(...)
```

to

```
OsgiBundleXmlApplicationContext ctx =
new OsgiBundleXmlApplicationContext(...)
```

In this change, the instance of the class `ResourceXmlApplicationContext` is replaced by an instance of the class `OsgiBundleXmlApplicationContext` to create a context object. The replacing class is provided by the `spring-osgi-core` library, which was not included in the ActiveMQ classpath. This Method Body Change involved the addition of the dependency to `spring-osgi-core` library in the `pom.xml` file denoting a build co-change.

```
<dependency>
 <groupId>org.springframework.osgi</groupId>
 <artifactId>spring-osgi-core</artifactId>
 ...
</dependency>
```

If the developer would not have updated the build file, the next Maven build would have failed due to the missing library. Testing the intra-project prediction models with only this work item, our models can predict the build co-change with an average Precision of 0.66.

The second example stems from the Karaf project in which a developer replaced the `Property` class from the `org.apache.felix.utils` library with the standard Java `Property` class (see the commit[8]). The two Method Body Changes and one Class Body Change involved an update of the `pom.xml` file in which the developer removed the dependency to the `org.apache.felix.utils` library. This change would not have led to an immediate build breakage, but in terms of having a clean project the removal of the dependency makes sense, for instance to minimize the chance of having class incompatibilities due to transitive dependency resolution.

Based on the analysis of the attribute importance of our prediction models and the manual analysis of build co-changes, we answer the third research question RQ3:

Number of Files, Method Body Changes, and Prior Build Co-Changes are the most important attributes for building intra- and cross-project prediction models, directly followed by the commit categories Management, Forward Engineering, and Reengineering.

VII. Threats to Validity

Regarding the validity of our results, we identify the following threats to the construct, internal, and external validity.

[6]https://www.atlassian.com/software/sourcetree

[7]http://tinyurl.com/glkgazw

[8]http://tinyurl.com/hm3oy6s

Construct Validity. A first threat to construct validity stems from our approach to aggregate commits to work items. We use search patterns to identify work items in the issue tracking systems and might miss linking commits to work items if they do not match our search patterns. We mitigate this issue by using bean-plots to compare the distribution of commits and work items over time selecting only projects that graphically show similar distributions, indicating that commits are adequately linked to work items (see Figure 2). This approach has also been used in the previous study by McIntosh *et al.* [4].

A second threat is due to the algorithms that we use to measure our new attributes to train and test our prediction models. We used Change Distiller to extract source code change categories that might not extract all possible changes in source files. Change Distiller has been intensively evaluated in previous studies [8], [28], which show that it covers most of the changes occurring in Java projects. Furthermore, Change Distiller handles changes that cannot be mapped by assigning them to the *Unclassified Changes* category. Regarding the commit categories, we use the heuristics of Hattori *et al.* [9] that have been evaluated with nine open source projects.

Internal Validity. One threat to internal validity relates to our selection of training and test sets. We address this threat by using repeated random sub-sampling, repeating the experiments 100 times with randomly selected training and test sets to minimize bias in our results.

A second threat is due to our selection of attributes to explain build co-changes. We might have overlooked other attributes that could help to explain build co-changes and may improve the prediction models. Similar to McIntosh *et al.* [4], we selected metrics that cover a wide range of change characteristics and added further attributes to describe source code change and commit categories.

Finally, several of our systems, such as Wicket, have a low build co-change ratio. Classifiers like random forest focus on correctly classifying the majority class because this yields better overall classification performance. We addressed this threat by investigating three state-of-the-art resampling methods that improved the performance of our prediction models.

External Validity. The main threat to external validity concerns the generalizability of our results, since we performed the experiments with ten Java open source projects that use Maven as their build tool. We mitigated this threat by selecting a range of different projects of different size comprising Java frameworks and end-user systems. However, further experiments with industrial systems, systems written in other programming languages and systems using other build tools instead of Maven are needed.

VIII. Conclusions

Software systems change and several of these changes require accompanying changes to the build system. A missing build co-change can lead to build breakage which costs time and money [3]. In this paper, we propose an improved model for predicting build co-changes. We extended the prior work of McIntosh *et al.* [4] with source code change and commit categories and achieved the following results:

- (RQ1) Source code change and commit categories significantly improve the model of McIntosh *et al.* [4] for intra-project prediction and the model of Xia *et al.* [5] for cross-project prediction.
- (RQ2) Resampling applied to better balance the training data set improves the models for intra- and cross-project prediction of build co-changes. In particular, projects with poor unrebalanced prediction performance benefit the most from rebalancing.
- (RQ2) SMOTE [10] and combinations with ENN [41] or TomekLinks [42] yield the best minority class improvement.
- (RQ3) *Number of Files*, *Method Body Changes*, and *Prior Build Co-Changes* are the most important attributes for building intra- and cross-project prediction models, directly followed by the commit categories *Management*, *Forward Engineering*, and *Reengineering*.

Future work. The proposed models are trained to predict possible build co-changes. We plan to extend this study to verify the study with a larger number of projects as well as with industrial projects to investigate a broader range of projects. Furthermore, we want to use the prediction models to predict the number of build co-changes and the category of the build co-change. This can give developers hints on where to start for build co-changing. We also plan to identify typical patterns of the co-evolution of source-/test code and the build system code. Furthermore, we plan to investigate improvements of our prediction models, such as by adding information from issue tracking systems.

References

[1] B. Adams, K. D. Schutter, H. Tromp, and W. D. Meuter, "The evolution of the linux build system," *Electronic Communication of the European Association of Software Science and Technology*, vol. 8, 2007.

[2] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge, "Programmers' build errors: a case study (at google)," in *Proceedings of the International Conference on Software Engineering*. ACM, 2014, pp. 724–734.

[3] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 41–50.

[4] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 241–250.

[5] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, "Cross-project build co-change prediction," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2015, pp. 311–320.

[6] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Proceedings of the International Conference on Program Comprehension*. IEEE, 2006, pp. 35–45.

[7] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, pp. 26–33, 2009.

[8] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[9] L. Hattori and M. Lanza, "On the nature of commits," in *Proceedings of the International Conference on Automated Software Engineering*. ACM/IEEE, 2008, pp. 63–71.

[10] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[11] G. Kumfert and T. Epperly, "Software in the doe: The hidden overhead of the build," *Lawrence Livermore National Laboratory, Technical Report*, 2002.

[12] L. Hochstein and Y. Jiao, "The cost of the build tax in scientific software," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2011, pp. 384–387.

[13] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of java build systems," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 578–608, 2012.

[14] ——, "The evolution of ANT build systems," in *Proceedings of the International Working Conference on Mining Software Repositories*. IEEE, 2010, pp. 42–51.

[15] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan, "A large-scale empirical study of the relationship between build technology and build maintenance," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1587–1633, 2015.

[16] X. Xia, D. Lo, X. Wang, and B. Zhou, "Build system analysis with link prediction," in *Symposium on Applied Computing*. ACM, 2014, pp. 1184–1186.

[17] B. Zhou, X. Xia, D. Lo, and X. Wang, "Build predictor: More accurate missed dependency prediction in build configuration files," in *Proceedings of the Annual Computer Software and Applications Conference*. IEEE, 2014, pp. 53–58.

[18] B. Adams, H. Tromp, K. D. Schutter, and W. D. Meuter, "Design recovery and maintenance of build systems," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 2007, pp. 114–123.

[19] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2012, pp. 650–660.

[20] J. M. Al-Kofahi, H. V. Nguyen, A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Detecting semantic changes in makefile build code," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 2012, pp. 150–159.

[21] R. Hardt and E. V. Munson, "An empirical evaluation of ant build maintenance using formiga," in *Proceedings of the International Conference on Software Maintenance and Evolution*, 2015, pp. 201–210.

[22] ——, "Ant build maintenance with formiga," in *Proceedings of the International Workshop on Release Engineering*. IEEE, 2013, pp. 13–16.

[23] A. E. Hassan and K. Zhang, "Using decision trees to predict the certification result of a build," in *Proceedings of the International Conference on Automated Software Engineering*. IEEE, 2006, pp. 189–198.

[24] J. Ratzinger, T. Sigmund, P. Vorburger, and H. C. Gall, "Mining software evolution to predict refactoring," in *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*. ACM/IEEE, 2007, pp. 354–363.

[25] W. M. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A. E. Hassan, "Should I contribute to this discussion?" in *Proceedings of the International Working Conference on Mining Software Repositories*. IEEE, 2010, pp. 181–190.

[26] P. Knab, M. Pinzger, and A. Bernstein, "Predicting defect densities in source code files with decision tree learners," in *Proceedings of the International Workshop on Mining Software Repositories*. ACM, 2006, pp. 119–125.

[27] D. Romano and M. Pinzger, "Using source code metrics to predict change-prone Java interfaces," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 2011, pp. 303–312.

[28] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proceedings of the International Working Conference on Mining Software Repositories*. ACM, 2011, pp. 83–92.

[29] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the International Conference on Software Engineering*. ACM, 2011, pp. 141–150.

[30] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proceedings of the International Symposium on Foundations of Software Engineering*. ACM, 2008, pp. 2–12.

[31] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[32] L. An, F. Khomh, and B. Adams, "Supplementary bug fixes vs. reopened bugs," in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 205–214.

[33] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2015, pp. 341–350.

[34] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[35] D. M. W. Powers, "Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation," *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37 – 63, 2011.

[36] P. Kampstra, "Beanplot: A Boxplot Alternative for Visual Comparison of Distributions," *Journal of Statistical Software*, vol. 28, pp. 1–9, 2008.

[37] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*. ACM, 2009, pp. 121–130.

[38] T. H. D. Nguyen, B. Adams, and A. E. Hassan, "A case study of bias in bug-fix datasets," in *Proceedings of the Working Conference on Reverse Engineering*. IEEE, 2010, pp. 259–268.

[39] R. J. Grissom and J. J. Kim, *Effect Sizes for Research: A Broad Practical Approach*. Lawrence Erlbaum Associates, 2005.

[40] H. He and E. A. Garcia, "Learning from imbalanced data," *Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.

[41] D. L. Wilson, "Asymptotic properties of nearest neighbor rules using edited data," *Transactions on Systems, Man, and Cybernetics*, vol. 2, no. 3, pp. 408–421, 1972.

[42] I. Tomek, "Two modifications of CNN," *Transactions on Systems, Man, and Cybernetics*, vol. 6, pp. 769–772, 1976.

[43] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *SIGKDD Explorations*, vol. 6, no. 1, pp. 20–29, 2004.