Using Structural Holes Metrics from Communication Networks to Predict Change Dependencies

Igor Scaliante Wiese[2], Rodrigo Takashi Kuroda[1], Douglas Nassif Roma Junior[2], Reginaldo Ré[2], Gustavo Ansaldi Oliva[3], and Marco Aurélio Gerosa[3]

[1] PPGI - UTFPR/CP
rodrigokuroda@gmail.com
[2] UTFPR/CM
nassifrroma@gmail.com, {igor,reginaldo}@utfpr.edu.br
[3] Department of Computer Science - IME/USP
{goliva,gerosa}@ime.usp.br

Abstract. Conway's Law describes that software systems are structured according to the communication structures of their developers. These developers when working on a feature or correcting a bug commit together a set of source code artifacts. The analysis of these co-changes makes it possible to identify change dependencies between artifacts. Influenced by Conway's Law, we hypothesize that Structural Hole Metrics (SHM) are able to identify strong and weak change coupling. We used SHM computed from communication networks to predict co-changes among files. Comparing SHM against process metrics using six well-known classification algorithms applied to Rails and Node.js projects, we achieved recall and precision values near 80% in the best cases. Mathews Correlation metric was used to verify if SHM was able to identify strong and weak co-changes. We also extracted rules to provide insights about the metrics using classification tree. To the best of our knowledge, this is the first study that investigated social aspects to predict change dependencies and the results obtained are very promising.

Keywords: structural holes metrics, social network analysis, change dependencies, communication network, Conway's law.

1 Introduction

A good understanding of the impact of communication and cooperation on software evolution can help researchers and practitioners to gain more insights to improve software quality [25,21]. According to some researchers [22,9,18], software modularity and quality can be associated with the interactions among developers. This phenomena was described by Conway, who states that "organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations" [11]. In this way, Conway's Law assumes a strong association between the software architecture and

N. Baloian et al. (Eds.): CRIWG 2014, LNCS 8658, pp. 294–310, 2014.
© Springer International Publishing Switzerland 2014

the communication structure [2]. During software evolution, developers communicate asynchronously in discussions related to change requests and cooperate around this requests changing files to fix bugs or add new features. The resulting changes may increase the source code complexity and disorganization [8] and are a social product [28]. For example Node.js (Table 2) used the constraintAVG (average of the lack of holes among neighbors) and hierarchySUM (sum of the concentration of constraints to a single node) six times. Considering Rails, neither of these metrics were frequently selected. For Rails (Table 3),effectiveSizeAVG (average of the portion of non-redundant neighbors of a node) and efficiencyAVG (average of the normalization of the effective size by the number of neighbors) – the others two SHM – were selected more often. Ball et al. [1] introduced the concept of co-changes (a.k.a. logical coupling and change dependencies) that involves the notion that some artifacts frequently change together during software development. Previous works have suggested that change dependencies in files influence the quality of software and indicated that entities that changed together in the past are likely to change together in the future [12]. Hence, finding insights of how these co-changes happen in software projects and how their social structures were organized can help developers planning, deciding, accommodating certain types of changes, and tracing through the effects of co-changes in software quality and architecture [20,9].

To understand the impacts of the communication structures, previous studies have been using graph-based analysis [27,4]. In fact, graph-based analysis have shown good results in several different works, for example, to predict defects, to estimate bug severity, to prioritize refactoring efforts, to predict defect-prone releases, and to understand socio-technical aspects [3,6,20,9,25].

Different types of metrics in Social Network Analysis (SNA) have been used to explore the graphs generated from software engineering data [27,3,6,20]. Among them, Structural Holes Metric (SHM) can reflect gaps between nodes in a social network indicating that a developer on either side of the hole have access to different flows of information [7,27]. However, to the best of our knowledge, SHM were studied only as part of social network metrics. Thus, as these metrics were merged with others in previous studies, like centrality and ego measures, their importance remains unclear.

Once that Conway's Law can explain the direct influence of the communication on the quality of software architecture, we expect that it also can explain the indirect relation between communication and co-changes. So, influenced by Conway's Law, we hypothesize that SHM has an important role to identify co-changes and can be explored to provide insights about the organization and the way that source code is modified. We claimed that SHM can describe the social organization during software changes. We used them to build classification models to predict strong and weak change coupling among files, whereas we find a lack of understanding about their prediction importance in previous works. Thus, two main questions were investigated: *RQ1. Does the SHM from communication can predict change dependencies? RQ2. Which is the role of SHM from communication to predict co-changes?*

We used four SHM (effective size, efficiency, constraint, and hierarchy) and three process metrics: sum of number of commits of each file coupled (we called "commits"), the number of updates that the co-change was committed (we called "updates"), and the number of distinct developers that committed each co-change (we called "developers"). To conduct this study we gathered data from two open source projects (Rails and Node.js) hosted on GitHub and built the communication networks using the issues of each project to calculate SHM. We used six well-known classification algorithms comparing SHM against process metrics and against all metrics together.

In summary, the main contributions of this study are:

- We found evidences that SHM computed from communication networks can be used to identify strong and weak co-changes;
- We shown that SHM was better than process metrics in some timeframes of analysis considering the two projects analyzed;
- We extracted a set of rules using classification tree that can provide insights of how communication can be related to co-changes.

The rest of the paper is organized as follows. In Section 2, we present related work about logical coupling and SNA. Section 3 shows the study design. We describe the process to collect the data, build the networks, compute the co-changes, and explain the metrics used in the classification approach. In Section 4, we answered RQ1 comparing the results of classification among each set of metrics using communication network, and RQ2 presenting the classification tree rules to compare the importance of each predictor. Section 5 discusses the threats to validity. Finally, conclusions and future work are presented in Section 6.

2 Related Works

Some related studies are presented in this section. First, we present the works related to logical coupling. Then, we discuss about previous works focused on SNA to build prediction models. We highlight that we only fond studies that predict co-changes without using SNA from communication networks and other set of papers that used SNA to predict only faults instead of co-changes.

2.1 Logical Coupling Studies

Some studies on logical coupling focuses on defects in open source projects [13,17]. D'Ambros et al. [13] performed a study on three large software systems (ArgoUML, Eclipse JDT, and Mylyn) and found that there was higher correlation between change coupling and defects than the one observed only with complexity metrics and defects. In turn, Kouroshfar [17] investigated the impact of specific characteristics of co-change dispersion on software quality. He used statistical regression models to show that co-changes that included files

from different subsystems resulted in more bugs than co-changes that include files only from the same subsystem.

Zimmermann et al. [30] developed a tool to try to avoid defects insertion during file changes. They built an Eclipse plug-in that collects information about source code changes from repositories and warns the developers about probable missing co-changes. They used association rules to suggest change coupling among files in method and file level. The authors reported precision values around 30% and recommended that for projects with continuous evolution of file changes the analysis should be made in file-level instead of method-level.

Zhou et al. [29] present a study similar to Zimmermann et al., since they proposed a model do predict co-changes. However, the model is more elaborate because it uses different predictors in Bayesian networks to predict the change coupling between source code entities. They extracted features like static source code dependency, past co-change frequency, age of change, author information, change request, and change candidate. They conducted experiments on two open source projects (Azureus and ArgoUML) and reported precision values around 60% and recall values of 40%. These last two papers are more related to our work. They predicted co-changes, but they did not use social metrics. We also tested more classification algorithm than Zhou et. al.

2.2 Social Network Analysis

Some studies on SNA used different types of networks to build prediction models. Wolf et al. [27] reported results indicating that developer communication plays an important role in software quality. They predicted build failure on IBM's Jazz project yielding recall values between 55% and 75%, and precision values between 50% to 76%. Bicer et al. [5] created models to predict defects on IBM's Jazz project and Drupal. Their results revealed that compared to other metrics such as churn metrics, social network metrics considerably decreased high false alarm rates. We considered that these works are more related to this study, whereas they are the only work that used SHM metrics from communication networks to build prediction models. However, these studies did not report the SHM individual performance and were used to predict fault instead of co-changes.

Other works, like Meneely et al. [19] examined the development network derived from code churn information. They conducted a case study and found a significant correlation between file-based development network metrics and failures. Bird et al. [6] evidenced the influence of combined sociotechnical software networks on the fault-proneness of individual software components. They reported results with precision and recall around 85%.

3 Study Design

In this section, we present how we collected the data, calculated the co-changes and the SHM, and built the classification models and evaluated the results.

3.1 Data Collection

We collected data from two open source projects: Rails and Node.js. We chose these projects because of their influential nature on GitHub[1] ecosystem [16]. We considered the number of "stars" and "forks" on GitHub indicating interesting projects. While Rails have more than 21.000 stars and 7.700 forks, Node.js have more than 28.800 stars and 6.100 forks. Table 1 presents some characteristics of each project collected from the Ohloh.net[2]. We present for each project the number of strong and weak co-changes and the percentage of imbalance in our dataset.

We noticed that the timeframes 2012.1 for Node.js and 2012.2 for Rails have the greater imbalance of our dataset. Class imbalance problem may lead to misclassification of the minority class. The classification algorithms normally are more focused on predicting the majority class [26]. For both projects, weak co-changes were the majority class for all timeframes. We observed that Node.js has more co-changes than Rails. The exception was observed on 2012.1, in which Rails and Node.js have similar number of co-changes considering both classes. Node.js presented a very high number of co-changes in 2011.2 compared to all other timeframes for both projects.

Table 1. Projects characteristics on different timeframes

		2011.1		2011.2		2012.1		2012.2	
		Strong	Weak	Strong	Weak	Strong	Weak	Strong	Weak
Node.js	# Co-changes	221	302	4208	5303	56	157	311	485
	% Imbalance	42	58	44	56	26	74	39	61
Rails	# Co-changes	12	19	47	56	65	138	66	193
	% Imbalance	39	61	46	54	32	68	25	75

We used the GitHubAPI[3] to gather data from the history of the source code and pull requests from the project. We grouped all pull requests and commit metadata considering six months as a timeframe, following the approach of Bird et al. [6]. Besides, we excluded pull requests without commits as we needed commits with at least two files.

To count the number of co-changes, we combined all files of the same pull request in pairs. Going through all pull requests, for each timeframe, we accumulated the number of times that a given pair of files occurred in the history of the project. We removed all pairs with less than five couplings in a timeframe. Five couplings were used as support count to remove a co-change from our analysis, the same approach adopted by Zimmermann et al. [30]. The support count determines the number of occurrences of pair of files in distinct pull requests.

[1] `https://github.com`

[2] `http://www.ohloh.net/` accessed on 10/04/2014.

[3] GitHubAPI can be access on: `http://developer.github.com/`

Since we are interested in using classification algorithms to predict how SHM describe the co-changes of files, we categorized the change dependencies into two classes. The first class represents strong change dependencies based on the amount of co-changes. The second class represents pair of files with weak change dependencies. To split the classes, we used the mean of total co-changes identified in each timeframe. Similar approach was used by Bird et al. [6] to study software defects.

3.2 Communication Networks and Structural Holes

A network is commonly represented as a graph G with a set of edges E and a set of nodes N. This structure is used to represent relationships observed from software development process and evolution [3]. In our network, each developer represents a node. Edges represent the exchange of messages between developers. We weighted the edges counting how many times each developer interacted with the others over the sequence of comments made in all pull requests where the co-changes were identified.

Suppose that developer 1 (d1) submitted a pull request and developer 2 (d2) made a comment. We create nodes d1 and d2 and connect d2 to d1 using a directed edge. Suppose that developer d3 write another message. We create the d3 node and connect it to d1 and d2. Before creating a node, we check if the node already existed. When creating an edge, we check if there is already an edge between two developers. If it is the case, we increment the weight value of this edge.

Burt [7] noticed that SHM concerns the notion of redundancy in networks and the degree to which there are missing links between nodes. SHM denote gaps between nodes in a social network or represents that people on either side of the hole have access to different flows of information, indicating that there is a diversity of information flow in the network. In the following, we describe four metrics proposed by Burt.

- **Effective Size** is the portion of non-redundant neighbors of a node. High values represent that many nodes among the neighbors are not redundant. Low values indicate that there is few non-redundant neighbors.
- **Efficiency** normalizes the effective size by the number of neighbors. High values show that many neighbors are non-redundant. Low values indicate a high redundancy among the neighbors.
- **Constraint** measure the lack of holes among neighbors. This measure is based on the degree of exclusive connection. Low values indicate that there is few alternative to access a single neighbor. High values indicate many alternatives to access the neighbors.
- **Hierarchy** measures the concentration of constraint to a single node. High values indicate that the constraint is concentrated in a single neighbor. Low values indicate that the constraint measure is the same for all neighbors.

Since we are interested in predicting co-changes using SHM obtained from communication networks, we needed to convert the SHM measures computed

from each developer for each co-change. To convert the SHM measures, we used the maximum, average, and sum to aggregate them. This approach was adapted from Meneely et al. [19].

3.3 Classification Approach

Classification algorithms learn from training sets composed of a set of features. In this study, the features represent properties of both files and interactions of developers and files. Our algorithms predict the change dependencies of a file, classifying them as "strong" or "weak." The algorithms read the training set and learn which features are most useful to differentiate the classes. We used SHM as features to different machine learning algorithms. We also computed the number of commits, number of updates, and number of developers that committed the file as predictors. Number of commits was computed as the sum of all times that each file was committed. Number of updates means the number of times that the pair was committed together. The number of developers was computed as the number of distinct developers that committed the co-change.

Typically, classification studies assess the predictive power of their model using recall, precision, and F-measure. This is a common way to evaluate the performance of prediction approaches and is widely used in related literature [24]. We calculated recall, precision, F-measure, and MCC for each timeframe and compare the results. We used the Confusion Matrix to compute these metrics. The values returned by the Confusion Matrix are True Positive (TP), True Negative (TN), False Negative (FN), and False Positive (FP) [23]. We calculated recall to identify the proportion of instances of a category that the model could successfully identify. We calculated precision to measure the percentage of correct identification in a class [23]. A good model yields both high precision and high recall. However, increasing one often reduces the other. F-measure returns a balanced score of recall and precision, calculated as the harmonic mean of these two metrics [23].

We also used the Matthews correlation coefficient (MCC) to show the quality of our predictions. MCC is a correlation coefficient between the observed and predicted classifications. This measure takes into account TP, FP, TN, and FN values and is generally regarded as a balanced measure, which can be used even if the classes are unbalanced. This measure returns a value between -1 and +1. A coefficient of +1 represents a perfect prediction, 0 means a random prediction and -1 indicates total disagreement between prediction and observations. We calculated the MCC by using the expression [23]:

$$MCC = \sqrt{\begin{array}{l}(((Recall(ClassStrong) + Recall(ClassWeak)) - 1) * \\ (((Precision(ClassStrong) + Precision(ClassWeak)) - 1)\end{array}}$$

We employed a 10-fold cross validation to evaluate the predictive power of the models. The cross-validation approach split the data, in our case a semester, into 10 sets, each containing 10% of the data. One set is used for testing data, and

the remaining nine sets are used as training data. Once we have class imbalanced dataset we used cross validation to generate 10 random folds to train and test our hypothesis. We report the average of the 10-fold cross validation. By building the models and predicting change dependencies, we aimed to investigate the influence of SHM in the prediction. We used the Orange tool [15] to create the learning models and evaluate them. The workflow and raw data can be downloaded from https://github.com/igorwiese/criwg2014.

4 Results

The main aim of this work was to verify if SHM computed from communication networks can be used to predict change dependencies. In the following sections, we discuss the results for each research question.

4.1 RQ1. Can SHM from Communication Networks Predict Change Dependencies?

To study change dependencies, we split them into two classes: strong and weak, following the procedures described in Section 3.1. We used the Orange Toolbox [15] and Minitab Statistical Software to analyze the data.

Following Rahman and Devanbu [24] recommendation, we used more than one classification algorithm to reduce the risk of depending on a particular learning technique. We applied 7 well-known classification approaches, namely: Classification Tree (CT); k-Nearest Neighbors (KNN); Neural Networks (NN); Support Vector Machine (SVM); Regression Logistic (RL); Naïve Bayes (NB); and an Ensemble Bagging (EB) combined with the best algorithm found in each timeframe.

Two experiments were conducted in order to respond RQ1. First, we used all metrics together in a single vector of features. Second, we compared each set of metric splitting the SHM and process metrics to evaluate the performance individually.

Table 2 presents the results for each timeframe for Node.js using SHM and process metrics together. We only reported F-measure, recall, precision, and MCC values for class "strong," because these dependencies are more important since they happen more often during each timeframe of analysis. We highlighted the best values of each evaluation metric in bold. This experiment was performed to check how the whole set of metrics could predict change dependencies.

The results show that we achieved high results of F-measure, precision, and recall at least in three of four periods. Considering 2011.1 using RL, we got 0.75, 0.78, and 0.73 respectively. For 2011.2, we achieved the best results with Neural network. The worst results comparing each semester was in 2012.1. Using NB, we got 0.50, 0.40, and 0.67. Using Bagging+NB, we got better values for precision, but we lost performance to precision. The same happened in 2012.2 using Bagging+KNN. For the last timeframe using KNN, we achieved 0.67, 0.67, and 0.65.

Table 2. Node.js Classification Results

	Cross Validation - 10 Fold															
	2011.1				2011.2				2012.1				2012.2			
	F1	Prec	Rec	MCC	F1	Prec	Rec	MCC	F1	Prec	Rec	MCC	F1	Prec	Rec	MCC
CT	0.73	0.73	0.74	0.54	0.61	0.65	0.58	0.34	0.30	0.34	0.26	0.09	0.64	0.67	0.62	0.43
KNN	**0.75**	0.75	**0.76**	0.57	0.60	0.63	0.58	0.31	0.47	0.45	0.50	**0.28**	**0.67**	0.67	**0.65**	**0.48**
NN	0.74	0.76	0.72	0.56	**0.65**	**0.70**	**0.61**	**0.41**	0.29	0.39	0.23	0.12	0.62	0.75	0.53	0.45
SVM	0.69	0.68	0.70	0.46	0.57	0.69	0.49	0.34	0.00	0.00	0.00	0.00	0.56	0.57	0.55	0.29
RL	**0.75**	**0.78**	0.73	**0.59**	0.61	0.69	0.54	0.37	0.33	0.45	0.26	0.18	0.60	0.69	0.52	0.40
NB	0.64	0.58	0.71	0.34	0.56	0.68	0.45	0.32	**0.50**	0.40	**0.67**	**0.28**	0.57	0.53	0.62	0.27
EB	**0.75**	**0.78**	0.71	0.58	0.61	0.69	0.55	0.37	0.35	**0.48**	0.28	0.21	0.62	**0.77**	0.52	0.47

We also reported the MCC values. As mentioned in Section 3.3, MCC metric means the correlation of the algorithms and features to predict both classes. For all timeframes, we achieved results ranging from 0.28 to 0.59. We can observe that for 2011.1, 2011.2, and 2012.2 we got good results showing moderate positive correlation to predict both classes. The worst result can be related to the high number of instances and the class imbalance problem noticed and reported in Table 1.

We observed that in each timeframe, we got better results using different classification algorithms. Trying to find the better algorithm for Node.js, we followed D'ambros et al. [14] recommendations to pairwise comparing classification algorithms. We used Mann-Whitney U test (at 95% confidence level) to conduct the comparisons. This test is non-parametric [10] and compare two group of means. The null hypothesis of this test indicates that the mean of the two groups are equal. This way, we grouped the results of F-measure and MCC for each algorithm for all timeframes. After this analysis, we could not reject the null hypothesis at 95% of significance. Thus, there is not a best algorithm because all of them had similar results.

We analyzed the variability of each algorithm using boxplot for both evaluation metrics. Figure 1 shows that NB and CT had the smaller variability among all algorithms. We observed that SVM algorithm presented the highest variability.

We performed the same analysis on Rails and we noticed similar results to the ones found in the Node.js project. Table 3 presents the results for Rails. We obtained the best results of F-measure, precision, recall, and MCC for 2011.1 using NB (0.69, 0.64, 0.75, and 0.47). For the second period, KNN got the best results (0.72, 0.70, 0.71, and 0.47). For 2012.1 and 2012.2, we got the best results using CT. However, we highlight that NB always achieved better values of recall considering all timeframes. The worst results were obtained for 2012.1.

We tried again to find the best algorithm by means of the Mann-Whitney U test (at 95% confidence level) comparing the mean of F-measure and MCC of each algorithm. Once again, we did not reject the null hypothesis.

Figure 2 presents the variability of each evaluation metric of the classification algorithms. Similar to Node.js, we found that CT and NB presented the smaller variability for both evaluation metrics. SVM obtained the highest variability of

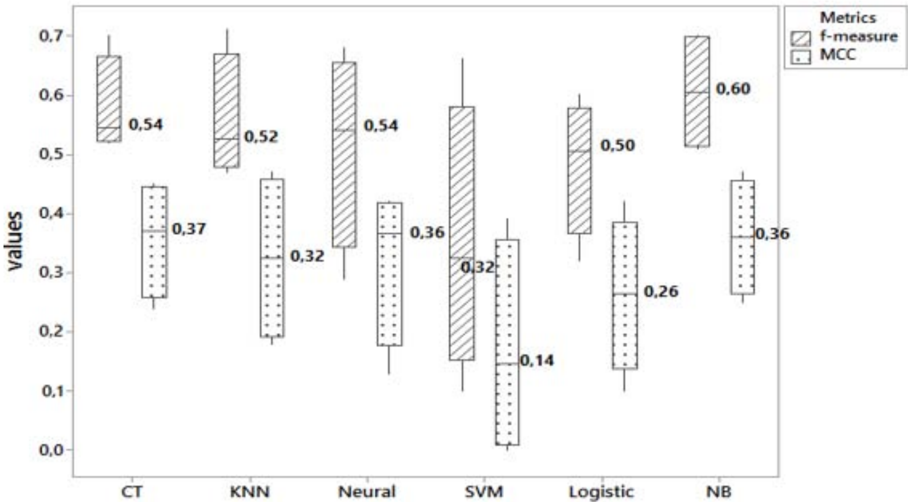

Fig. 1. Boxplot of F-measure and MCC for Node.js

Table 3. Rails Classification Results

	Cross Validation - 10 Fold															
	2011.1				2011.2				2012.1				2012.2			
	F1	Prec	Rec	MCC	F1	Prec	Rec	MCC	F1	Prec	Rec	MCC	F1	Prec	Rec	MCC
CT	0.52	0.54	0.50	0.24	0.70	0.69	0.72	0.45	**0.53**	0.52	0.55	0.31	**0.56**	0.61	0.53	**0.43**
KNN	0.50	0.50	0.50	0.18	**0.71**	0.70	0.72	**0.47**	0.47	0.47	0.47	0.23	0.55	0.62	0.50	0.42
NN	0.58	0.58	0.58	0.32	0.68	0.68	0.68	0.42	0.29	0.46	0.21	0.13	0.50	**0.70**	0.39	0.41
SVM	0.31	0.42	0.25	0.04	0.66	0.67	0.65	0.39	0.0	0.0	0.0	0.0	0.34	0.57	0.24	0.25
RL	0.50	0.62	0.41	0.28	0.60	0.59	0.61	0.25	0.32	0.41	0.26	0.10	0.51	0.69	0.40	0.42
NB	**0.69**	**0.64**	**0.75**	**0.47**	0.70	0.63	**0.78**	0.41	0.52	0.45	**0.61**	0.25	0.51	0.41	**0.68**	0.31
EB	0.64	0.61	0.66	0.39	0.67	**0.71**	0.63	0.42	0.52	**0.58**	0.47	**0.33**	0.51	0.63	0.43	0.40

F-measure and MCC for Rails. Once we achieved good results for both projects using all metrics together, we performed a more fine-grained analysis to answer RQ1 comparing each set of metrics independently. To perform this analysis, we split SHM and process into two groups of metrics. We also randomly selected only three classification algorithms, since we did not find statistical difference among them.

Comparing the results of F-measure and MCC of each set of metrics, we found that SHM had better results on the last semester (2012.2), and process metrics got better results on the first semester. Considering the second and third timeframes, the results were similar. Due to space constraints, we will not show the complete summarization of these results, but we provide an addendum with the results on our GitHub repository (https://github.com/igorwiese/criwg2014).

To statistically compare the set of metrics, we used Mann-Whitney U test (at 95% confidence level). Once again, we could not rejected the null hypothesis, showing that both set of metrics had similar performance to predict change dependencies. Figure 3 presents the results of each evaluation metric for both

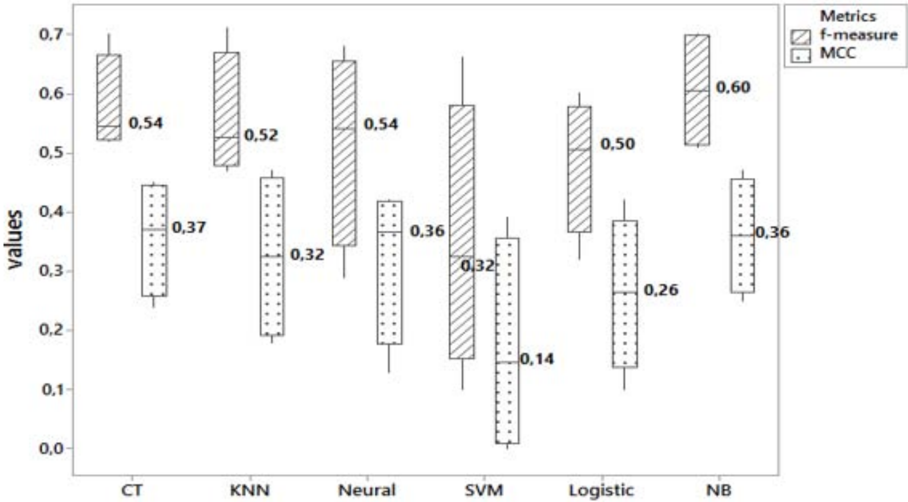

Fig. 2. Boxplot of F-measure and MCC for Rails

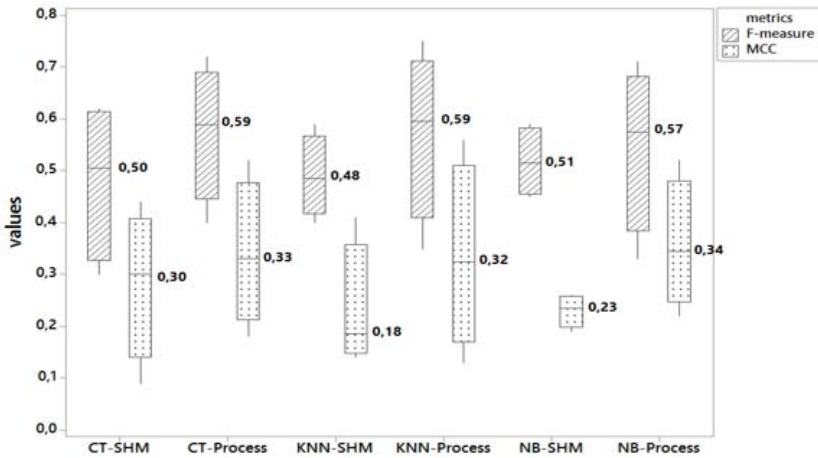

Fig. 3. Comparison among set of metrics and algorithms for Node.js

SHM and process metrics for Node.js. We observed that NB and KNN had the smaller variability using SHM metrics. However, the best result was achieved using KNN+process metrics.

Performing the same analysis for Rails, we achieved good results using SHM on 2011.1. For the other three semesters, we observed a small difference among the set of metrics. We did not find statistical difference between SHM and process metrics. We also present the boxplot to provide insights of how each set of metric works. Figure 4 presents the results for Rails. We can observe that the variability

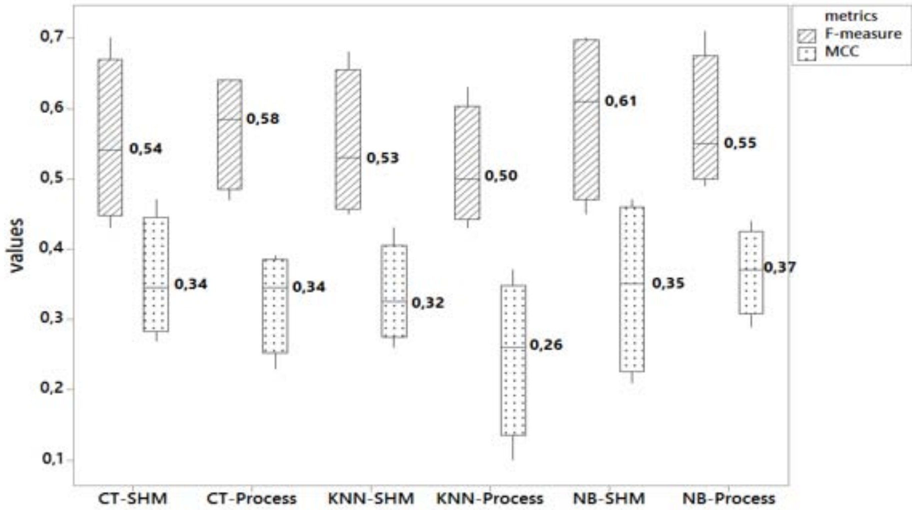

Fig. 4. Comparison among the set of metrics and algorithms for Rails

among the algorithms and set of metrics are very similar. The best results were obtained using CT and NB with SHM metrics.

As we reported in this section, for both projects, SHM and process metrics can be used to predict change dependencies. Together they can improve the results. However, they also can be used as individual predictors without losing predictive power in terms of recall, precision, F-measure, and MCC.

4.2 RQ2. Which Is the Role of SHM from Communication to Predict Co-changes?

To provide insights about how each metric help to predict change dependencies, we used the Orange tool [15] to mine rules from classification tree. We started the analysis building the classification tree for both projects using only SHM, because we did not find statistical difference among the set of metrics and we were interested in investigating the prediction performance of SHM. Then, we analyzed the results combining both set of metrics together.

Figure 5 presents the CT for Node.js for 2011.1 timeframe. The root metric represents the most relevant metric. In this case, constraintAvg was the most relevant metric. Using constraintAvg higher than 0.503 and smaller or equal than 0.503, we split the tree in two parts. The much closer from the root of the tree, the most important the metric is. Red squares means that we found a rule to describe "strong" class. The blue square means that "weak" class was described. For example, we found the rule "IF constraintAVG $>$ 0.503 and hierarchySUM $>$ 2.594 THEN class="strong" to the red square on the third level on the left side. We can observe that using this rule, 27 instances can be predicted as "strong" coupling. Considering the right side, we showed two examples of rules using

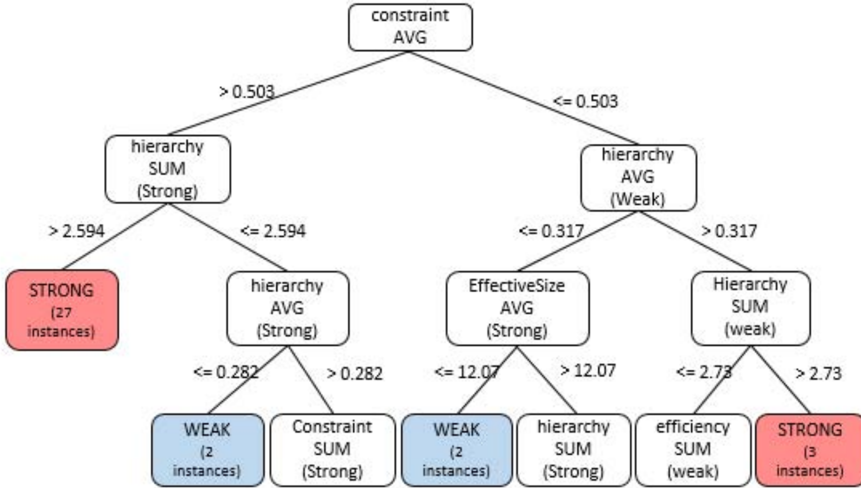

Fig. 5. Classification Tree for Node.js 2011.1

hierarchyAVG on the second and third level consecutively: (i) constraintAVG <= 0.503 and hierarchyAVG <=0.317 THEN "strong," (ii) constraintAVG > 0.503 and hierarchySUM <= 2.594 and hierarchyAVG > 0.282 THEN "strong."

To rank the metrics we counted the amount of times that each metric appeared on the first four levels of each tree for both projects. For Node.js, we ranked the metrics following this order: constraintAVG (6), hierarchySUM (6), hierarchyAVG (3), efficiencySUM (3), constraintMAX (2), constraintSUM (20), effectiveSizeSUM (2), effectiveSizeAVG (1), and efficiencyAVG (1). Considering Rails, we ranked the metrics following this order: effectiveSizeAVG (3), efficiencyAVG (3), effectiveSizeSUM (2), effectiveSizeMAX (1), efficiencySUM (1), constraintSUM (1), constraintAVG (1), and hierarchyAVG (1).

Conducting these analyses, we found that all SHM were used, however "effective size" were most frequently selected. Comparing Rails with Node.js, we noticed that Rails needed the less amount of SHM to predict change dependencies than Node.js. We also investigated the classification tree using all metrics together. We ranked all metrics following this order to Node.js project: commits (19), updates (14), effectiveSizeSUM (6), efficiencySUM (2), constraintMAX (3), hierarchyMAX (2), efficiencyAVG (2), constraintAVG (2), hierarchyAVG (1), and hierarchySUM (1). For Rails project, we can ranked the metrics following this order: commits (8), updates (3), efficiencyAVG (3), effectiveSizeSUM (3), effectiveSizeAVG (2), constraintSUM (2), efficiencySUM (2), efficiencyMAX (1), hierarchySUM (1), hierarchyAVG (1), and constraintAVG (1).

The root position on the classification tree is the most important metric. We found two process metrics at this position (commits and updates) and one SHM (constraintMAX - 2 times) considering Node.js. For Rails, we observed similar results, finding two process metrics (commits and updates) and two SHM (effectiveSizeAVG and efficiencySUM).

Only to present some examples, we choose three rules mined from classification tree used on Node.js project to predict change dependency. This three rules were mined from communication network for 2011.1 timeframe: (i) IF commits <= 86.0 AND updates <=7.5 AND effectiveSizeSUM <=115.64 THEN class="weak", (ii) IF efficiencySUM > 1.31 AND hierarchyAVG <=0.32 AND updates in (12.5,28.5] THEN class="strong", and (iii) IF efficiencySUM > 1.31 AND hierarchyAVG > 0.32 AND updates in (12.5,28.5] THEN class="strong".

5 Threats to Validity

Co-changes identification: We used five pull requests as support count to remove co-changes of our study. Although this threshold is used in the literature, we still need to evaluate the impact of other values.

Timeframe selection: We used six months to identify the co-change among files. Furthermore, we used "five pull requests" as support to remove pair of files that were committed four or less times among this six months of timeframe. Other approaches like association rules was used to find co-change using a snapshot from source code history. We plan to study the impact of this technique on the GitHub repository to better identify logical coupling and determine what is the best timeframe to group co-changes and build the communication networks. We also planned to run the experiments considering each Release instead of a fix six months timeframe.

Generalizability: This study aimed to explore the use of SHM to predict change dependencies. We used two popular projects from GitHub, but additional studies are necessary to generalize the results to other projects. This is our first effort to predict change dependencies and we plan to add new social network metrics and more projects to investigate the impact of others social metrics over change dependencies.

6 Conclusion and Future Works

During software evolution, developers communicate and cooperate changing files and discussing about change requests. A good understanding of the impact of communication and cooperation can provide insights to improve software quality. Supported by Conway's law that states about the impact of communication over the software design, we used SHM computed from communication networks to study change dependencies. We computed four different SHM, to predict strong and weak change dependencies among files changed in pull request submissions.

We answered RQ1 investigating if all of SHM and process metrics were useful to build classification models, since we did not found previous works that explored social aspects to predict change dependencies. Using seven well-known classification algorithms, we achieved similar results of recall, precision, f-measure, and MCC metrics compared to previous works that used social network metrics to predict bugs and works that predict co-changes using non-social metrics. We did not find statistical difference among algorithms for our two set

of metrics (SHM vs process metrics). However, we found that structural metrics for Rails project had a small advantage since it presented smaller variability than process metrics. For Node.js, the results among the two sets were similar. We shown this results presenting boxplots for each project.

Since we found that SHM could be used to predict change dependencies, we explored which of the four metrics were more relevant. We inspected the rules used to build the classification tree and predict strong and weak couplings. We found that Node.js needed more different types of metrics than Rails. These two projects presented very different rules, for example Node.js (Table 2) used the constraintAVG (average of the lack of holes among neighbors) and hierarchySUM (sum of the concentration of constraints to a single node) six times. Considering Rails, neither of these metrics were frequently selected. For Rails (Table 3), effectiveSizeAVG (average of the portion of non-redundant neighbors of a node) and efficiencyAVG (average of the normalization of the effective size by the number of neighbors) – the others two SHM – were selected more often. We observed that different metrics were selected in specific timeframes.

As main contribution, we shown that SHM obtained from communication networks can support the Conway's law and predict change dependencies. Since this first effort to use social metrics presented promising results, we will explore more broadly this social dimension of development process. We want to investigate in future works additional projects and social metrics, and go deeper in providing insights for managers and developers about how their collaboration patterns can influence code quality. Besides, as we are looking into relations of social interaction (communication) and technical work (cooperation), one can investigate the application of these metrics in other collaboration domains.

Acknowledgments. We thank Fundação Araucária, NAWEB and NAPSOL for the financial support. Marco G. receives individual grant from CNPq and FAPESP. Igor W. and Igor S. receive grants from CAPES (Process BEX 2039-13-3).

References

1. Ball, T., Adam, J.K., Harvey, A.P., Siy, P.: If your version control system could talk. In: ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering (March 1997)
2. Betz, S., Mite, D., Fricker, S., Moss, A., Afzal, W., Svahnberg, M., Wohlin, C., Borstler, J., Gorschek, T.: An evolutionary perspective on socio-technical congruence: The rubber band effect. In: RESER 2013, pp. 15–24 (2013)
3. Bhattacharya, P., Iliofotou, M., Neamtiu, I., Faloutsos, M.: Graph-based analysis and prediction for software evolution. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 419–429 (June 2012)
4. Biçer, S., Bener, A.B., Çağlayan, B.: Defect prediction using social network analysis on issue repositories. In: Proceedings of the 2011 International Conference on Software and Systems Process, pp. 63–71 (2011)

5. Bicer, S., Bener, A.B., Cauglayan, B.: Defect prediction using social network analysis on issue repositories. In: Proceedings of the 2011 International Conference on Software and Systems Process, ICSSP 2011 (2011)
6. Bird, C., Nachiappan, N., Harald, G., Brendan, M., Devanbu, P.: Putting it all together: Using socio-technical networks to predict failures. In: Proceedings of the 2009 20th International Symposium on Software Reliability Engineering (2009)
7. Burt, R.: Structural Holes: The Social Structure of Competition. Harvard University Press (1995)
8. Canfora, G., Cerulo, L., Cimitile, M., Di Penta, M.: How changes affect software entropy: an empirical study. Empirical Software Engineering 19(1), 1–38 (2014)
9. Cataldo, M., Mockus, A., Roberts, J.A., Herbsleb, J.D.: Software dependencies, work dependencies, and their impact on failures. IEEE Transactions on Software Engineering 35(6), 864–878 (2009)
10. Conover, W.J.: Practical Nonparametric Statistics, vol. 2. John Wiley and Sons
11. Conway, M.E.: How do committees invent? Datamation (1968)
12. D'Ambros, M., Lanza, M.: Reverse engineering with logical coupling. In: 13th Working Conference on WCRE 2006, pp. 189–198 (2006)
13. D'Ambros, M., Lanza, M., Robbes, R.: On the relationship between change coupling and software defects. In: WCRE 2009, pp. 135–144 (2009)
14. D'Ambros, M., Lanza, M., Robbes, R.: Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empirical Softw. Eng. 17(4-5), 531–577 (2012)
15. Demšar, J., Curk, T., Aleš, E., Gorup, Č., Hočevar, T., Mitar, M., Martin, M., Matija, P., Marko, T., Anže, S., Miha, Š., Lan, U., Lan, Ž., Jure, Ž., Marinka, Ž., Blaž, Z.: Orange: Data mining toolbox in python. Journal of Machine Learning Research 14, 2349–2353 (2013)
16. Ferdian, T., Tegawende, F.B., Lo, D., Jiang, L.: Network structure of social coding in github. In: 15th European Conference on Software Maintenance and Reengineering, pp. 323–326 (2013)
17. Kouroshfar, E.: Studying the effect of co-change dispersion on software quality. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013, pp. 1450–1452 (2013)
18. Kwan, I., Cataldo, M., Damian, D.: Conway's law revisited: The evidence for a task-based perspective. IEEE Software 29(1), 90–93 (2012)
19. Meneely, A., Williams, L., Will, S., Osborne, J.A.: Predicting failures with developer networks and social network analysis. In: SIGSOFT FSE, pp. 13–23 (2008)
20. Meneely, A., Williams, L., Snipes, W., Osborne, J.: Predicting failures with developer networks and social network analysis. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT 2008/FSE-16, pp. 13–23. ACM, New York (2008)
21. Nagappan, N., Murphy, B., Basili, V.R.: The influence of organizational structure on software quality: an empirical case study. In: International Conference on Software Engineering (ICSE), pp. 521–530 (2008)
22. Panichella, S., Canfora, G., Di Penta, M., Oliveto, R.: How the evolution of emerging collaborations relates to code changes: An empirical. In: IEEE International Conference on Program Comprehension (ICPC 2014) (2014)
23. Powers, D.M.W.: Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation. Technical Report SIE-07-001, School of Informatics and Engineering, Flinders University, Adelaide, Australia (2007)
24. Rahman, F., Devanbu, P.T.: How, and why, process metrics are better. In: ICSE, pp. 432–441 (2013)

25. Tsay, J., Dabbish, L.A., Herbsleb, J.D.: Influence of social and technical factors for evaluating contribution in github. In: ICSE (2014)
26. Visa, S.: Issues in mining imbalanced data sets - a review paper. In: Proceedings of the Sixteen Midwest Artificial Intelligence and Cognitive Science Conference, pp. 67–73 (2005)
27. Wolf, T., Schroter, A., Damian, D., Nguyen, T.: Predicting build failures using social network analysis on developer communication. In: Proceedings of the 31st International Conference on Software Engineering, ICSE 2009 (2009)
28. Zanetti, M.S., Sarigöl, E., Scholtes, I., Tessone, C.J., Schweitzer, F.: A quantitative study of social organisation in open source software communities 28, 116–122 (2012)
29. Zhou, Y., Wursch, M., Giger, E., Gall, H., Lu, J.: A bayesian network based approach for change coupling prediction. In: 15th Working Conference on WCRE 2008, pp. 27–36 (2008)
30. Zimmermann, T., Zeller, A., Weissgerber, P., Diehl, S.: Mining version histories to guide software changes. IEEE Transactions on soft Engineering 31(6), 429–445 (2005)

View publication stats