# Change Prediction through Coding Rules Violations

Irene Tollin
Alten Italia
Milano, Italy
irene.tollin@alten.it

Francesca Arcelli Fontana, Marco Zanoni, Riccardo Roveda
Department of Informatics, Systems and Communication,
University of Milano-Bicocca
Milano, Italy
(arcelli,marco.zanoni,riccardo.roveda)@disco.unimib.it

## ABSTRACT

Static source code analysis is an increasingly important activity to manage software project quality, and is often found as a part of the development process. A widely adopted way of checking code quality is through the detection of violations to specific sets of rules addressing good programming practices. SonarQube is a platform able to detect these violations, called Issues. In this paper we described an empirical study performend on two industrial projects, where we used Issues extracted on different versions of the projects to predict changes in code through a set of machine learning models. We achieved good detection performances, especially when predicting changes in the next version. This result paves the way for future investigations of the interest in an industrial setting towards the prioritization of Issues management according to their impact on change-proneness.

## CCS CONCEPTS

• **Software engineering → Software evolution.**; *Software maintenance tools.*;

## KEYWORDS

software quality, change prediction, issues, machine learning.

**ACM Reference format:**
Irene Tollin and Francesca Arcelli Fontana, Marco Zanoni, Riccardo Roveda. 2017. Change Prediction through Coding Rules Violations. In *Proceedings of EASE'17, Karlskrona, Sweden, June 15-16, 2017,* 4 pages.
DOI: http://dx.doi.org/10.1145/3084226.3084282

## 1 INTRODUCTION

The assessment of software quality is an important aspect of the lifecycle of software projects. To support automatic and semi-automatic software quality assessment, a large amount of code metrics can be extracted, and many other indicators useful to assess quality can be detected.

This work describes the result of a collaboration between the Laboratory of Software Evolution and Reverse Engineering[1] and the Alten company in Milano (Alten is an international company[2] leader in engineering and technology consulting). The goal of this collaboration has been to relate quality measures extracted by means of static analysis to the change-proneness of projects' elements. Specifically, we employed machine learning techniques to predict changes in the projects using the quality measures extracted over time onthem. The study involves software quality analysis, considering both static code analysis and version control system analysis.

To perform the experiments, different tools and platforms have been used. The study has been applied to two industrial projects developed by Alten, two web applications that use C# for the back-end and JavaScript for the front-end. The first project (called Project A in the following) has a life time of 7 months and has been created to monitor inbound and outbound call center performances; the second one (Project B) has a life time of 14 months and has been created to report results about customer satisfaction. Both projects are developed in close collaboration with the customers, following their continuous feedback, by 4 developers in a team composed by 8 people. The code of both projects is maintained in a Team Foundation Server (TFS) version control repository, and is analyzed through SonarQube (SQ), to extract *Issues*, i.e., violations to coding rules and/or metric thresholds associated with bad quality.

In this paper, we focus in particular on this Research Question (RQ): *To what extent can Issues extracted with SQ be used to predict changes?* The answer to this RQ will help us to understand if the Issues are actually linked to properties of the code that make it more change-prone and in particular, give a quantitative estimation of the practical relevance of the considered Issues.

We started this collaboration between University and Industry, since SonarQube is largely used worldwide by software development companies. We were interested in starting to analyze if the identified issues by SQ are really worth of attention/improvement, i.e., if SQ really identifies problems that are urgent for the developers to fix in the near future. We choose this analysis strategy exploiting machine learning techniques for two reasons. First, we deal with many different quality indicators (SQ issues) at once, making dependency analysis through hypothesis testing or correlation coefficients very inefficient. Second, machine learning models are actionable, e.g., to understand if prediction performances are stable across different projects or not.

## 2 RELATED WORK

We briefly introduce some work in the area of change prediction. Code Churn is a popular measure when dealing with change prediction. It is also used, e.g., as a predictor of post-release failures [8]. More recently, changes have been related with static code quality analysis, e.g., Romano et al. [11] use antipatterns to predict source

[1] http://essere.disco.unimib.it
[2] http://www.alten.it/


*EASE'17, Karlskrona, Sweden*
 978-1-4503-4804-1/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3084226.3084282

**Table 1: Issues Severities**

| Severity | Description |
|---|---|
| Blocker | Code must be fixed immediately |
| Critical | Code must be immediately reviewed |
| Major | Quality flaw which can highly impact the developer productivity |
| Minor | Quality flaw which can slightly impact the developer productivity |
| Info | Neither a bug nor a quality flaw, just a finding |

code changes finding that classes affected by antipatterns change more frequently; Malhotra and Khanna [6] use search-based techniques and software metrics to predict changes in object-oriented software. The importance of identifying change-prone components also led to the analysis of change-proneness in Service-Oriented Architectures [10]. In this paper, we do not use or predict churn (number of added or changed lines of code), but we more coarsely rely on file change information, for both learning and prediction.

## 3 BACKGROUND

In the following, we define basic terms and concepts used in the paper regarding our main source of information, i.e., SonarQube.

SonarQube[3] is an open source platform for continuous inspection of code quality [3]. It supports more than 20 languages[4] through different plugins, the computation of many code metrics, and the possibility to check the conformance to a large number of coding rules. Violations to coding rules are called *Issues*. These Issues will be exploited as input data to predict changes in the analyzed projects. To conduct this experiment, we adopted the default set of Issues managed by SonarQube. Issues extracted from SQ are divided in 5 importance levels, known as severities, spanning from Blocker to Info[5]. Table 1 reports severities and their definition. Issues are also coarsely classified in three categories: *Bugs* are bad programming practices known to represent mistakes in most cases, *Vulnerabilities* are bad programming practices that can lead to a security Issue or unwanted behaviour, while *Code smells* generally refers to Issues leading to bad code comprehension, e.g., high size, high complexity (this category is only weakly related to the known code smells of Fowler [4]).

## 4 EXPERIMENT SETUP

We extracted SQ Issues and churn metrics on different project versions, and subsequently integrated the two sources of information. The analyzed versions have been selected as follows: For Project A, which started in January 2016, we selected a version for every week, while for Project B we selected a version every two weeks.

### 4.1 Issues Extraction

Issues have been extracted using SQ v5.5, through the available plugins supporting C# and JavaScript. Since we analyzed SQ results outside the platform, we exported all data in external tables through SQ's APIs (there are no export facilities in the platform). SQ keeps a full record of all extracted data only for the last version of an analyzed project. For this reason, we analyzed all projects versions as different projects (using the version number as a suffix for project's name). Each project version has been prepared before the analysis, by setting different analysis parameters[6], i.e., the name and key of the project (necessary to distinguish between versions), the base directory of the project, source encoding, paths to be excluded from the analysis (the projects' folders contained third party libraries), programming languages used in the project.

Given the particular nature of C# projects, the property files also contained specific references to the project "solution" `.sln` file, containing the build data used by MSBuild in the project. This ensured the use of the recommended scanner for MSBuild projects[7].

After the analysis, we exported all Issues found by SQ in Excel files. The exported files report, for each detected Issue (each Issue is on a different row):

- Project Key: string distinguishing the project versions;
- File in which the Issue has been detected;
- Line of the file in which the Issue exists;
- Severity of the Issue;
- ID of the Issue (represent the rules violated by this Issue).

**Table 2: CodeMaat data excerpt for "Project A"**

| Entity | Added | Deleted | N. Commits |
|---|---|---|---|
| BundleConfig.cs | 5 | 1 | 2 |
| filterManager.js | 45 | 5 | 1 |
| site.js | 15 | 1 | 2 |
| adminManager.js | 13 | 4 | 3 |
| dataTableManager.js | 1 | 0 | 1 |
| SmsService.cs | 138 | 5 | 3 |
| DashboardContextService.cs | 4 | 11 | 1 |
| WarRoomService.asmx.cs | 24 | 0 | 1 |

### 4.2 Change Extraction

The second source of information are the changes between the analyzed versions. In particular, we extracted churn measures. Since the projects are maintained on TFS, but most freely available tools for software repository analysis are available for Git, we first transformed the two TFS repositories in Git repositories through a tool called git-tfs[8]. Then we applied another tool, called CodeMaat [9], to extract code churn metrics between each subsequent pair of versions of the two projects. The metrics extracted by CodeMaat are: the number of Added lines, of Deleted lines and of Commits made on each file. The results of this process is a set of `.txt` files equal to the number of versions of the projects analyzed. An example of the output (simplified) is reported in Table 2: for each file, the metric values are reported in the respective columns.

[3] http://www.sonarqube.org/
[4] http://docs.sonarqube.org/display/PLUG/Plugin+Library
[5] http://docs.sonarqube.org/display/SONAR/Issues
[6] http://docs.sonarqube.org/display/SONAR/Analysis+Parameters
[7] http://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+for+MSBuild
[8] https://github.com/git-tfs/git-tfs
[9] https://github.com/adamtornhill/code-maat

**Table 3: Issues - Code Churn join from Project A v18**

| Prj | File | Version | Add | Del | NoC | Line | Issue | Severity |
|---|---|---|---|---|---|---|---|---|
| A | DataService.cs | 18 | 5 | 2 | 1 | 490 | S2360 | MAJOR |
| A | WSmsService.cs | 18 | 3 | 3 | 1 | 37 | S3457 | MINOR |
| A | adminManager.js | 18 | 2 | 2 | 1 | 135 | S1854 | MAJOR |
| A | Dashboard.aspx.cs | 18 | 25 | 5 | 1 | 48 | S1854 | MAJOR |

Add=Added, Del=Deleted, NoC=Number of Commits

## 4.3 Data Preparation

After extracting our data describing Issues and Churn, we needed to integrate them before starting the machine learning experiment. In this phase, we relied on Knime, the Analytics Platform [12] which provides a large set of data manipulation and analysis features. For each version of each project, we had a `.xls` file containing Issue information and a `.txt` file containing Churn information. We obtained for each project a single dataset listing all Issues in all files of all versions of the projects, and the Churn measures associated to each file at the specific version. During the process, we also cleaned the data coming from CodeMaat, excluding the files that were not subject of analysis in SQ, i.e., external libraries included in the projects. An excerpt of the output dataset is shown in Table 3, which refers to the 18th version of Project A.

## 4.4 Experimental method and performance measures

We define our experiment as a supervised classification task. It is the task of learning a target function $f$ that maps each attribute set $x$ to one of the predefined class labels $y$. Among the techniques used to calculate a classifier's accuracy, we selected 10-fold cross-validation [9]. Every cross-validation experiment produces a confusion matrix that is used to describe the performance of a classifier model on a set of (unseen) test data for which the true values are known. In the matrix, the actual values can be either True or False (T/F) and predictions can be Positive or Negative (P/N), generating four possible combinations: TP, TN, FP, FN. We consider the standard performance measures, derived from the confusion matrix: Accuracy, Recall, Precision and F-measure.

We selected different classifiers to be used in our experiment: Decision Trees [7], Naive Bayes Networks [5] and Random Forests [5].

# 5 CHANGE PREDICTION RESULTS

We performed our learning experiment using two different setups, called *timelines* in the following, and two different representation for the Issues.

The first timeline uses the information about the presence of Issues in the past 4 versions to predict any change in the next 2 versions (we refer to this as *Prediction Model* in the following). In the training phase we need to provide to the classifiers both the features (the Issues) and the labels (changes). Therefore, the data points used to train the classifiers contain, for each version of the project, change information about the 2 previous versions and Issue information about the 4 versions preceding the latter. If we number the current version with 0, each data point contains changes for versions {-1,0} and Issues for versions {-5, -4, -3, -2} (we refer to this as the *Training Model* in the following). The training dataset is built by shifting these six-version window over all the available versions of the project.

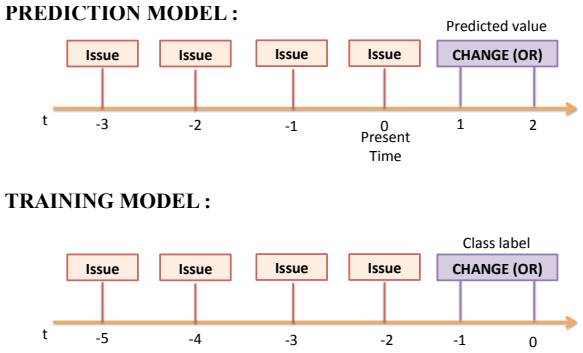


**Figure 1: Timeline 1**

**Table 4: Classification using binary Issue representation**

| | Model | Prj | TP | FP | Recall | Prec. | F-meas. | Acc. |
|---|---|---|---|---|---|---|---|---|
| Timeline 1 | Decision Tree | A | 111 | 59 | 0.163 | 0.653 | 0.261 | 0.609 |
| | | B | 1787 | 485 | 1 | 0.787 | 0.881 | 0.787 |
| | Random Forest | A | 374 | 116 | 0.549 | 0.763 | 0.639 | 0.737 |
| | | B | 271 | 0 | 0.559 | 1 | 0.717 | 0.906 |
| | Naive Bayes | A | 439 | 271 | 0.645 | 0.618 | 0.631 | 0.681 |
| | | B | 285 | 0 | 0.588 | 1 | 0.74 | 0.912 |
| Timeline 2 | Decision Tree | A | 80 | 35 | 0.169 | 0.696 | 0.272 | 0.734 |
| | | B | 320 | 0 | 0.66 | 1 | 0.795 | 0.927 |
| | Random Forest | A | 186 | 65 | 0.392 | 0.741 | 0.513 | 0.781 |
| | | B | 364 | 0 | 0.751 | 1 | 0.857 | 0.947 |
| | Naive Bayes | A | 280 | 339 | 0.591 | 0.452 | 0.512 | 0.669 |
| | | B | 376 | 0 | 0.775 | 1 | 0.873 | 0.952 |

All features have been modeled as binary features, represented using integer values 0 (false) and 1 (true). On each data point, Issues are represented as different features, each one named using the Issue ID and its relative position in the window, e.g., "Issue01 (-2)". Each feature is true if detected on the file in the respective version, false otherwise. To model changes, instead, we first represent the changes in the two versions as binary features (true if file has changed, false otherwise), and then we compose the two features in a single one through an OR operation. In this way, the Change label is true if the file received at least a change in one of the two versions. The representations of the Prediction Model and the Training Model for the first timeline are outlined in Figure 1. The performances obtained by the selected training models by means of cross-validation are reported in Table 4.

Classification accuracy is very high on Project B, with maximum precision. This is a good result, since with high precision (while recall is not high) the amount of false positives is low (or zero in this case). This is important, given the tendency of static analysis tools

to provide false positives or not-useful results. Moreover, since we are also interested in the long term in understanding which Issues are mostly related to change-proneness, these trained models can be analyzed more easily to extract that kind of relations. On Project A, performances are lower, and in particular precision is significantly lower than in Project B. As for the models, Random Forests and Naive Bayes have similar performances, while the ones for Decision Trees are lower.

The second timeline uses Issue information of the past 3 versions to predict changes in the next version. Therefore, the Training Model used Issues in versions {-3, -2, -1}, and Changes in version 0. The modeling of Issues is the same as the first timeline, while Changes simply consider a single version, and do not need the OR composition used in the first timeline. The performance obtained for this timeline are also reported in Table 4. In this case, performances distribution between the two projects and the three models is similar to the first timeline, but performances are generally higher, and in particular with better recall. Overall, we can notice the existence of a *locality* principle, i.e., Issues in the near past better explain changes in the near future.

## 6 THREATS TO VALIDITY

Threats to internal validity of our experiment are concerned with the data preparation and extraction. In our experiment, we use SQ to extract Issues from source code, that could have accuracy problems. This is highly improbable, since the definition of Issues is precise, leading to a deterministic Issue extraction, that can be subjected only to implementation bugs. Another source of possible problems in Issue data is the setup of projects. We payed attention in setup of SQ analysis, to configure only the project's source code and excluding code imported from libraries or not used. As for change information, we rely on an external tool, which parses git change logs, and after different tests, proved to be reliable. Another threat to the validity of our results can be seen in the dataset balance. In fact, each combination of projects and timeline has a different balance: Project A is more balanced, with 40% of positive instances in Timeline 1 and 30% in Timeline 2, while in Project B balances are 20% and 35% respectively. While the imbalance is not extreme, in future work we plan to introduce resampling techniques to further improve prediction performances. Threats to external validity are related to the generalization of results. The scope of our experiment is tied to industrial projects, implemented in C# and JavaScript, developed by the same company. The two projects have different goals but are structurally similar, being both web applications. For these reasons, our results may not be transferable to projects in other domains, developed by other companies or in other technologies.

## 7 CONCLUSIONS AND FUTURE WORK

This work is a result of an empirical study carried out on 2 projects developed by Alten. The study relates information extracted through static analysis, i.e., SQ Issues, to information about changes in the code. In particular, we applied machine learning to predict changes in files using Issues and their evolution as input information. We exploited classification models to predict changes, and experimented with different settings w.r.t. the way we use historical information.

After performing our experiments, the answer to our RQ (*To what extent can Issues extracted with SQ be used to predict changes?*) is: Issues are a *good predictor* for changes in files in our dataset; in particular, the use of *recent* Issues (the ones contained in the past three versions) to predict *near-future* changes (the ones in next version) give the best results. We obtained high accuracy ($\geq$0.9) and precision (=1) in particular for Project B, while in Project A performances are not so good, but yet lead to useful results (accuracy $\sim$ 0.78, precision $\sim$ 0.74). Given the good prediction performances, we can argue that code quality retrieved through static analysis and its evolution is actually tied to change-proneness. This makes quality analysis useful when trying to manage changes in the project and to obtain a better overall maintainability.

In future work, we plan to extend this kind of study to assess which Issues, and in which evolution patterns, are more tied to changes in the code. This information can be highly useful in the context of prioritizing [1] the most important Issues to be addressed in a system, and to filter out completely the Issues that do not provide objective benefits to the projects when they are removed. We would like to extend this study to larger projects using different technologies and belonging to different domains. As for the methodological aspects, we also plan to operationalize the experiment to make the data extraction and analysis automatic and scalable to larger data sets. This will also allow us to optimize the learning models' parameters according to the prediction performances. Moreover, we are interested to extend our study on change prediction, by considering also architectural issues or smells detected in a system [2].

## REFERENCES


[1] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Riccardo Roveda. 2015. Towards a Prioritization of Code Debt: A Code Smell Intensity Index. In *Proc. 7th Intern. Work.on Managing Technical Debt (MTD 2015)*. IEEE, Germany. DOI: https://doi.org/10.1109/MTD.2015.7332620

[2] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, and Marco Zanoni. 2016. Automatic Detection of Instability Architectural Smells. In *Proc. 32nd Intern. Conf. Soft. Maint.Evol.(ICSME 2016)*. IEEE, Raleigh, North Carolina, USA.

[3] G. Ann Campbell and Patroklos P. Papapetrou. 2013. *SonarQube in Action* (1st ed.). Manning Publications Co., Greenwich, CT, USA.

[4] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA.

[5] Sotiris B Kotsiantis. 2007. Supervised machine learning: A review of classification techniques. *Informatica* 31, 3 (2007), 249–268.

[6] Ruchika Malhotra and Megha Khanna. 2016. An exploratory study for software change prediction in object-oriented systems using hybridized techniques. *Automated Software Engineering* (2016), 1–45. DOI: https://doi.org/10.1007/s10515-016-0203-0

[7] Thomas M. Mitchell. 1997. *Machine Learning* (1 ed.). McGraw-Hill, Inc., New York, NY, USA, Chapter Decision Tree Learning.

[8] Nachiappan Nagappan and Thomas Ball. 2007. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proc. 1st Intern. Symp.Emp. Soft. Engin. and Measurement (ESEM 2007)*. IEEE, Madrid, Spain, 364–373. DOI: https://doi.org/10.1109/ESEM.2007.13

[9] Payam Refaeilzadeh, Lei Tang, and Huan Liu. 2009. *Cross-Validation.* Springer US, Boston, MA, 532–538. DOI: https://doi.org/10.1007/978-0-387-39940-9_565

[10] D. Romano. 2013. Analyzing the change-proneness of service-oriented systems from an industrial perspective. In *Proc. 35th Intern. Conf. on Software Engineering (ICSE 2013)*. IEEE, San Francisco, CA, USA, 1365–1368. DOI: https://doi.org/10.1109/ICSE.2013.6606718

[11] D. Romano, P. Raila, M. Pinzger, and F. Khomh. 2012. Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes. In *Proc. 19th Working Conf. Reverse Engineering (WCRE 2012)*. IEEE, Kingston, Ontario, Canada, 437–446. DOI: https://doi.org/10.1109/WCRE.2012.53

[12] Rosaria Silipo and Michael P Mazanetz. 2012. *The KNIME cookbook.* KNIME Press, Zürich, Switzerland.