A Multi-source Machine Learning Approach to Predict Defect Prone Components

Pasquale Ardimento[1], Mario Luca Bernardi[2] and Marta Cimitile[3]
[1]*Computer Science Department, University of Bari Aldo Moro, Via E. Orabona 4, Bari, Italy*
[2]*Department of Computing, Giustino Fortunato University, Benevento, Italy*
[3]*Unitelma Sapienza, Rome, Italy*

Keywords: Machine Learning, Fault Prediction, Software Metrics.

Abstract: Software code life cycle is characterized by continuous changes requiring a great effort to perform the testing of all the components involved in the changes. Given the limited number of resources, the identification of the defect proneness of the software components becomes a critical issue allowing to improve the resources allocation and distributions. In the last years several approaches to evaluating the defect proneness of software components are proposed: these approaches exploit products metrics (like the Chidamber and Kemerer metrics suite) or process metrics (measuring specific aspect of the development process). In this paper, a multi-source machine learning approach based on a selection of both products and process metrics to predict defect proneness is proposed. With respect to the existing approaches, the proposed classifier allows predicting the defect proneness basing on the evolution of these features across the project development. The approach is tested on a real dataset composed of two well-known open source software systems on a total of 183 releases. The obtained results show that the proposed features have effective defect proneness prediction ability.

1 INTRODUCTION

Add new features and/or increase the software quality require much effort invested to perform software testing and debugging. However, software developers resources are often limited and the schedules are very tight. Defect prediction can support developers and reduce the resources consumption through the identification of the software components that are more prone to be defective. Basing on the above considerations, several defect prediction machine learning classification approaches (Radjenović et al., 2013) are proposed in the last years. They are based on a classifier that predicts defect-prone software components basing on a set of features (i.e, code complexity, number of code lines, number or modified files) (Isong et al., 2016). Basing on the more recent studies (Radjenović et al., 2013), several machine learning based approaches adopt product metrics like Chidamber and Kemerer's (CK) (Chidamber and Kemerer, 1994; Boucher and Badri, 2016) object-oriented metrics as features. Even if these products metrics are surely highly correlated to defects as several studies confirms (Hassan, 2009), there are also other aspects, related to the adopted development process, that need to be taken into account to capture the complex technical and social implications of the phenomenon of bug introduction in a predictive model. In this paper a new machine learning approach to predict defect prone components is proposed. Differently, from the other approaches, the proposed classifier uses as features the evolution over releases of a mix of products and process metrics.

Section 2 discusses the related work and the backgrounds. Section 3 introduces the proposed classification process while Section 4 presents the evaluation of the proposed features effectiveness referring to their capability to evaluate the software code proneness and discusses the results of the evaluation. Finally, paper conclusions and future work are reported in Section 5.

2 RELATED WORK

The defect prediction approaches proposed in literature apply data mining and machine learning algorithms to classify the software components into defective or non-defective. These approaches utilize

Ardimento, P., Bernardi, M. and Cimitile, M.
A Multi-source Machine Learning Approach to Predict Defect Prone Components.
DOI: 10.5220/0006857802720279
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 272-279
ISBN: 978-989-758-320-9

a classification model obtained from the analysis of the earlier projects data considered as features (Lessmann et al., 2008). According to the literature (Boucher and Badri, 2016), several approaches are based on the adoption of CK metrics suite introduced in (Chidamber and Kemerer, 1994). Two experiments on the relation between object-oriented design metrics and software quality are proposed in (Basili et al., 1996; Briand et al., 2000). In (Kanmani et al., 2007), the effectiveness of the CK metrics for software fault prediction is evaluated by using two neural network based models. Similarly, multiple linear regression, multivariate logistic regression, decision trees and Bayesian belief network are used respectively in (Nagappan et al., 2005; Olague et al., 2007; Gyimothy et al., 2005; Kapila and Singh, 2013). In (Lessmann et al., 2008) a framework for comparative software prediction is proposed and tested on 22 classifiers and 10 NASA datasets. The obtained results show an encouraging predictive accuracy of the classifiers and no significant difference in the classifiers performances. Finally, in (Kaur and Kaur, 2018), the six better (basing on the previous studies) classification algorithms are used to compare models for fault prediction on open source software and compare the results of open software projects with an industrial dataset. The analysis of the literature shows that most studies on software defect prediction are focused on a comparative analysis of the available machine learning methods. In this paper, we propose a features model that takes into consideration the software component's history to consider the evolution of a mix of products and process metrics. The novelty resides in the application of an ensemble learning approach using the selected set of products and process metrics evaluated at different observation points in space (i.e. the project's classes) and time (i.e. at each release). This allows training a classifier that becomes more effective as the software project matures (i.e. more releases are generated) since more training data is available.

3 THE METHODOLOGY

The key aspects of our approach are the granularity of prediction (i.e. the code elements selected as the targets of the prediction), the metrics exploited as features, the adopted machine learning prediction techniques.

For that concerns the granularity, the proposed approach aims at identifying the defects at a class level. This means evaluating the defect proneness of a class across software project releases events. Most of the existing defect proneness prediction approaches are able to predict the defects only at source file level (Hassan, 2009; Moser et al., 2008) or, even worse, at the component level (Nagappan et al., 2010). Coarse-grained predictions force developers to spend effort in locating the defects. Since finer granularity, instead, can mitigate such issue, the proposed model has been designed to predict the defects at class level thus effectively restricting the testing effort required.

For what concerns features selection, there are two classes of metrics that are widely used to predict bugs: product metrics, that characterize features of the source code elements; process metrics, that characterize the development history of the source code elements. To take into account all these aspects when predicting defects, the proposed approach exploits a mix of the first two classes of features.

The last key aspect is the prediction technique exploited in our approach. In literature, machine learning techniques have been successfully exploited for bug prediction. These approaches(Moser et al., 2008), mostly adopting supervised learning, seems to suggest that the impact of the adopted metrics on the classification performances is much higher than the specific machine learning algorithm chosen to perform the classification. To confirm this, in our study several classification techniques based on supervised learning are compared and results discussed.

The overall defect proneness classification process is summarised in Figure 1 and the remaining of the section delves into the details of each step.

3.1 Data Collection

The process starts with the data collection step represented in Figure 1-(a) that extracts the information from the source code repository and the bug tracking system and integrates them into a single coherent dataset. The required information is spread between the issue tracker (BTS) and the source code repository (VCS). Such integration is centered on recovering the traceability links among the two systems to identify the commits making up each release and linking the issues reported in BTS to them.

To this regards a tool, written in Java, has been developed to execute the following tasks, for a given software project: downloads the VCS repository; evaluates the total number of releases, commits and the total number of commits for each release; calculates, for each release, all the revisions (commit IDs) that are related to it; collects information about commit date, commit author, commit URL, and other structural information. Then two scripts have been developed to automate the extraction process. The first

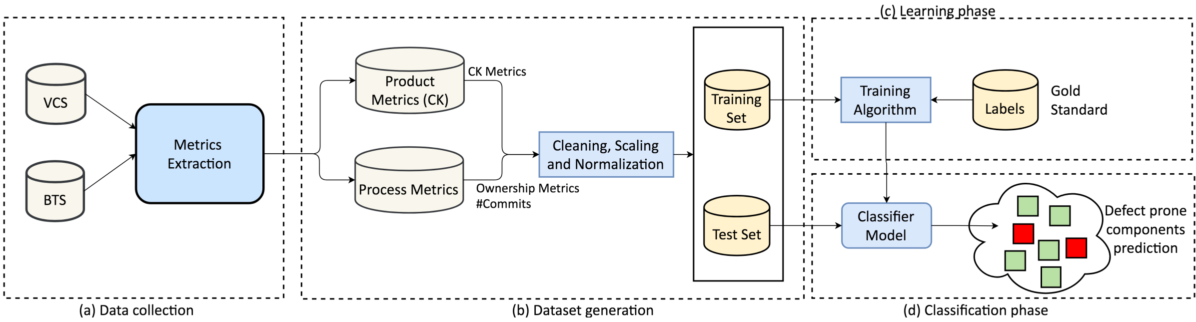

Figure 1: An overview of the defect prone components prediction process.

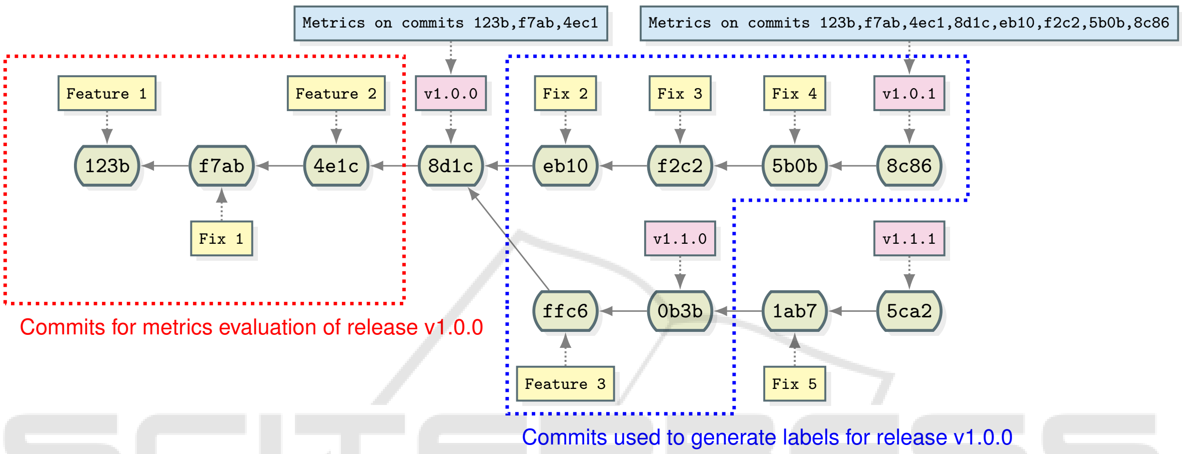

Figure 2: A small excerpt of a typical repository graphs, decorated with information on Issues, Bugs and Fixes.

script builds automatically all the release source codes, using Apache Ant or Apache Maven, while the second one evaluates the metrics of different classes and stores the results. A broad overview of the integration is depicted on Figure 2. A key problem in evaluating the metrics is to define the portion of the source code repository graph that is used for each release. Our approach, given the commit of a release (e.g. commit ID `8d1c` in the figure), creates a set of commits that, for that release, is built by following back the parent/children relationship of the repository DAG until to the branch root (this set is highlighted in red in Figure 2). For each commit found on this path, information gathered from BTS are considered in order to evaluate required metrics. Specifically, as Figure 1 shows, from these integrated data sources, two classes of defect predictors (product and process metrics) are generated and are used, later in the process, as features for the subsequent learning step.

3.1.1 Product Metrics

We selected a subset of the Chidamber-Kemerer (CK) ObjectOriented (OO) Metric Suite(Chidamber and Kemerer, 1994), reported in Table 1. To evaluate these metrics, the CKJM tool developed by Spinellis(Spinellis, 2005) was integrated into our toolchain to perform the experimentation. This tool is able to calculate Chidamber and Kemerer object-oriented metrics for each class of the system, by processing the bytecodes of compiled Java files. This means that, for each release, a complete snapshot of the entire source code must be extracted and compiled to produce the bytecodes feeding the CKJM tool.

3.1.2 Process Metrics

Since there are several factors, related to the development process with which the software project is developed, influencing the complex phenomenon of bug introduction. For this reason, process metrics are also taken into account looking at both the kind of changes that are performed and the properties of committers performing such changes. Table 2 reports the process metrics considered as features. To evaluate the number of commits, issues, and bugs for each class we build the log from the VCS repository and extract, for each commit, the files changed, the commit ID, the commit timestamp, the committer ID, and the commit note. With this information, we are also

Table 1: Product Metrics adopted as features to train the classifier.

Metric	Description
WMC	Weighted Method per Class - It measures the complexity of a class calculated by the cyclomatic complexities of its methods
DIT	Depth of Inheritance Tree - It calculates how far down a class is declared in the inheritance hierarchy.
NOC	Number of Children - It calculates how many sub-classes are going to inherit the methods of the parent class.
CBO	Coupling Between Objects - It evaluates the coupling between objects.
RFC	Response for a Class - The number of methods that can be invoked in response to a message in a class.
LCOM	Lack of Cohesion in Methods - It measures the amount of cohesiveness present.
CA	Afferent Couplings - It evaluates the number of classes that depend upon the measured class
NPM	Number of Public Methods - It evaluates all the methods in a class that are declared as public

Table 2: Process Metrics adopted as features to train the classifier.

Metric	Description
#Commits	The number of total commits on the class.
#Issues	The number of total issues (Improvements, Refactoring, and Tasks) involving the class.
#Bugs	The number of total bugs involving the class.
#Owner Committers	The number of owners that have worked on the class.
#Other Committers	The number of other committers (i.e. committers not owning the class) that have worked on the class.
#Commits by Others	The number of total commits on the class performed by committers not owning the class.

able to identify fix-inducing changes using an approach inspired by the work of Fischer (Fischer et al., 2003), i.e., we select changes with the commit note matching a pattern such as *bug ID*, *issue ID*, or similar, where *ID* is a registered issue in the BTS of the project. Hence the issue ID acts as traceability link between VCS and BTS systems. We then use the issue type/severity field to classify the issue and distinguish bug fixing changes from different kind of issues (e.g., improvement, enhancements, feature additions, and refactoring tasks). This is needed to compute, for each class, the process metrics (the number of improvement/refactoring issues and the number of fixes). Finally, we only consider issues having the status *CLOSED* and the resolution *FIXED* since their changes must be committed in the repository and applied to the components in the context of a release. Basically, we distinguish among: (*i*) issues that were related to bugs, since we use them as a measure of fault-proneness, and (*ii*) issues that are related to improvement and changes.

To identify fix-inducing changes, we use a modified version of SZZ algorithm presented in (Kim et al., 2008). It is based on the git *blame* feature which, for a file revision, provides for each line the revision of the last change occurred in time. More precisely, given the fix for the bug identified by k, the approach proceeds as follows:

1. let $p(c)$ be the parent of the commit c; for each file f_i, $i = 1 \ldots m_k$ involved in the bug fix k (m_k is the number of files changed in the bug fix k) and fixed in a commit with ID $R_{fix_{i,k}}$, we get the same file in its parent commit with ID $p(R_{fix_{i,k}})$, since it is the most recent revision that still contains the bug k.
2. starting from the revision with commit ID $p(R_{fix_{i,k}})$, the *blame* command of git is used to detect, for each line in f_i changed in the fix of the bug k, the commit, in the past, where the last change to that line occurred. Hence for each file f_i, we obtain a set of $n_{i,k}$ fix-inducing commit IDs $R_{b_{i,j,k}}$, $j = 1 \ldots n_{i,k}$.

The revisions $R_{b_{i,j,k}}$, $j = 1 \ldots n_{i,k}$ are finally used to perform bugs and issues counting aggregating by file and by release in order to evaluate the a subset of process metrics (i.e. the number of commits, fixes and bugs for each class in each release).

However further information is required to evaluate metrics related to ownership (i.e. the number of owners committers, the number of other committers and the number of commits on each class performed by the other committers).

In (Bird et al., 2011) authors defined a *file owner* as a committer that performed, on a file, a given percentage of the total number of commits occurred on that file. Hence we tag as owners the set of the committers of a file that collectively performed at least half of the total commits on that file.

More formally, we identify as the set of file owners for file f_i at time t the set:

$$O(f_i,t) \equiv \{o_1,\ldots,o_i\} \quad (1)$$

of committers that, overall, performed the 50% of commits on f_i in the time interval $[0,t]$ (where 0 is the starting point of our period of observation), i.e.,

$$\sum_{j=1}^{|O(f_i,t)|} C(o_j) \geq 0.5 \cdot T_c(f_i,t) \quad (2)$$

where $C(o_j)$ is the number of commits performed by o_i on file f_i in the time interval $[0,t]$, and $T_c(f_i,t)$ is the total number of commits performed on f_i in the time interval $[0,t]$. To calculate the set $O(f_i,t)$, we build an ascending sorted list of committers based on the number of commits they performed on f_i in the time interval $[0,t]$, and selected the minimum set of topmost committers in the ranked list that satisfies the above constraint.

By collecting all these metrics for each class of the system, at a given release, the feature vector reported in Figure 3 is constructed and used for training.

3.2 Dataset Generation Steps

The goal of this step, reported in Figure 1-(b), is to produce a dataset from the metrics evaluated from data sources that are suitable to be used to train a classifier adopting a supervised machine learning approach. The labeled match data consists of all the commits performed on the fixed file and the corresponding values of the metrics. The labeled match data is firstly cleaned (by removing the incomplete and wrong data) and normalized.

The cleaning consists of polish data produced during the pre-processing step in order to: i) fix missing values; ii) remove noise; iii) remove special character or values; iv) verify semantic consistency.

After the cleaning, the normalization is performed by using a Min-max scaling approach realizing a linear transformation of the original data. More precisely, if min_X is the minimum value for the attribute X and max_X is the maximum for X, the min-max normalization approach relates a value v_i of X to a v'_i in the range $\{newMin_X, newMax_X\}$ by computing:

$$v'_i = \frac{v_i - min_X}{max_X - min_X}(newMax_X - newMin_X) + newMin_X$$

From the normalized data, a dataset is derived. It contains the training set used to train the classifier and the test set used to make the performance assessment of the classifier.

3.3 Learning Phase

To perform the learning phase is necessary to calculate the labels (i.e. the defective components of releases used for training the classifier). Given a release X, we construct such labels by looking at the future commits until we reach on each subsequent branch the next releases. This allows considering fixes that apply to release X and from them recover, as already discussed, the fix-inducing changes. Files affected by fix-inducing changes are finally labeled as *DEFECTIVE* or *NOT-DEFECTIVE*.

The training is performed using a stratified K-fold cross-validation (Rodriguez et al., 2010) approach. It consists to split the data into k folds of the same size in which the folds are selected so that each fold contains roughly the same proportions of class labels. Subsequently, k iterations of training and validation are performed and, for each iteration, a different fold of the data is used for validation while the remaining folds are used for training. As pointed out, to ensure that each fold is representative, the data are stratified prior to being split into folds. This model selection method provides a less biased estimation of the accuracy.

3.4 Classification Phase

In this step, the trained classifier is ready to be used to perform prediction on new data extracted from the project. It could be applied to the live software project data and can be used to predict, at the current release, what are the classes that are the most likely to produce bugs until the next release cycle. Since the approach works on metrics that can be automatically extracted, it is suitable to be integrated into existing BTS or continuous integration (CI) platforms.

4 EVALUATION

4.1 Experiments Description

To validate the approach, an experiment was performed involving the following two projects, both written in Java Language: **Apache CXF**, a well known open source services framework, and **Apache Commons IO**, a widespread library that contains several facilities for text and binary data management. The dataset for Apache CXF contains 134 releases (from release 2.1 up to release 3.2.0), while the dataset for Apache Commons IO contains 49 releases (from release 1.0.0 up to release 2.5.0).

The experimentation was performed by using the classifiers, reported in Table 3, based on traditional

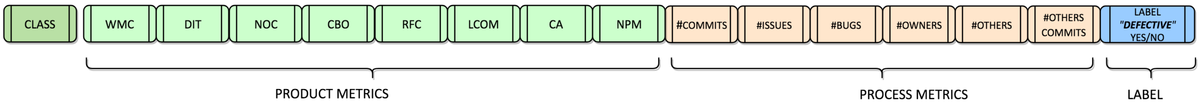

Figure 3: The features vector used to train the classifier.

Table 3: Machine-learning classification algorithms.

Classification Alghoritm	Description
J48	A decision tree is created based on the attribute values of the training dataset in order to classify a new item. The set of items discriminating the various instances is extracted so when a new item is encountered it can be classified.
Decision Stump	It consists to create a decision tree model with one internal root node immediately connected to the terminal nodes. The prediction is allowed by a decision stump basing on the value of just one input feature.
Hoeffding Tree	An incremental, anytime DT induction algorithm learning from massive data streams. Basing on the assumption that the distribution generating examples does not change over time, it often uses a small sample to choose an optimal splitting attribute.
Random Forest	It operates by constructing, at training time, a multitude of DTs and outputting the class that is the mode (the most frequent value appearing in a set of data) of the classes of the individual trees.
Random Tree	It constructs a tree containing some randomly chosen attributes at each node. No pruning is performed.
REP Tree	It builds a DT using information gain/variance and reduces errors using reduced-error pruning. The values are ordered for numeric attributes once and missing values are recovered with by splitting the corresponding instances into pieces.

Table 4: Best Precision, Recall and F-Measure for Apache CXF for the adopted learning algorithms.

Algorithm	P	R	F	Class
J48	0.82	0.88	0.85	Defective
	0.83	0.89		Not defective
Decision Stump	0.79	0.78	0.78	Defective
	0.81	0.71		Not defective
Hoeffding Tree	0.86	0.89	0.89	Defective
	0.88	0.91		Not defective
Random Forest	0.87	0.88	0.87	Defective
	0.87	0.90		Not defective
Random Tree	0.85	0.81	0.83	Defective
	0.83	0.86		Not defective
REPTree	0.80	0.81	0.80	Defective
	0.77	0.72		Not defective

Table 5: Best Precision, Recall and F-Measure for Apache Commons-IO for the adopted learning algorithms.

Algorithm	P	R	F	Class
J48	0.71	0.77	0.70	Defective
	0.74	0.80		Not defective
Decision Stump	0.68	0.69	0.66	Defective
	0.70	0.60		Not defective
Hoeffding Tree	0.78	0.81	0.82	Defective
	0.80	0.83		Not defective
Random Forest	0.76	0.77	0.78	Defective
	0.75	0.81		Not defective
Random Tree	0.74	0.72	0.74	Defective
	0.76	0.75		Not defective
REPTree	0.71	0.72	0.68	Defective
	0.66	0.64		Not defective

Machine Learning (ML) algorithms: J48, DecisionStump, HoeffdingTree, RandomForest, RandomTree and REPTree. A description of these algorithms is also reported in the same table. They all use a decision tree as a predictive model. The observations about an item are represented as the branches of the tree and they are used to obtain conclusions about its class (the leaves of the tree).

4.2 Evaluation Strategy

To validate the classifier we used the following classification quality metrics: Precision, Recall, F-Measure and ROC Area.

Precision has been computed as the proportion of the observations that truly belong to investigated class (i.e., defect proneness components) among all those which were assigned to the class (analyzed components). It is the ratio of the number of records correctly assigned to a specific class to the total number of records assigned to that class (correct and incorrect ones):

$$Precision = \frac{tp}{tp + fp} \tag{3}$$

where *tp* indicates the number of true positives and *fp* indicates the number of false positives.

The recall has been computed as the proportion of observations that were assigned to a given class, among all the observations that truly belong to the class. It is the ratio of the number of relevant records retrieved to the total number of relevant records:

$$Recall = \frac{tp}{tp + fn} \tag{4}$$

where *tp* indicates the number of true positives and *fn* indicates the number of false negatives.

The ROC Area is defined as the probability that a positive instance randomly chosen is classified above a negative randomly chosen. In the evaluation, a ROC curve for each release of a given system is estimated to study the performance of the classifiers as the version number increases (e.g. as the software project becomes more mature and more training data is available). Specifically, we expect that as history data size increases, the performance of the trained classifier consequently improves. Even if this is not always consistent across all the minor versions, this trend, as reported in the evaluation, is confirmed by the results of our study.

4.3 Discussion of Results

Tables 4 and 5 contain the results of classification process performed with the six algorithms of Table 3 for the two systems. The best performing algorithm in our experimentation resulted to be Hoeffding Tree on CXF scoring the best precision of 0.86/0.88 with a recall of 0.89/0.91 on the two classes and a ROC area of 0.89. Random Forest was on both system the second best algorithm experimented but, with respect to Hoeffding Tree, was much slower during training.

To study classification performances with respect to project age in terms of releases, the set of ROC curves were evaluated by release. Figure 4 shows an excerpt of the 134 ROC curves for Apache CXF that highlight what we expected: as history becomes larger the AUC of the classifier improves and the classification becomes more reliable (in our experimentation the peak in the accuracy is obtained at 0.89 for release 3.1.6).

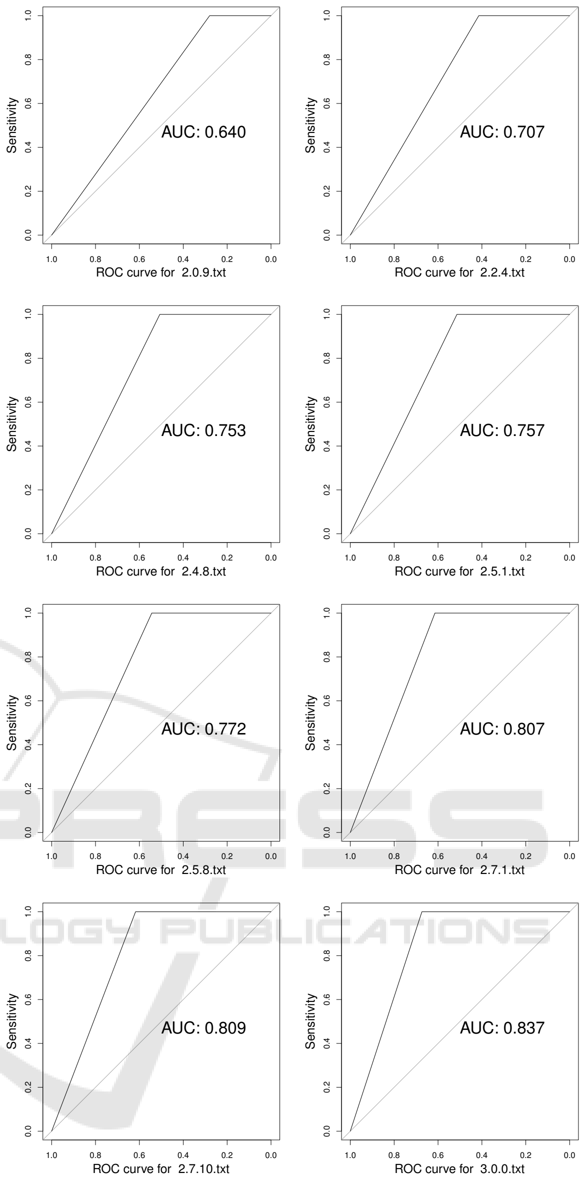

Figure 4: The plots of estimated ROCs for 16 versions of Apache CXF from 2.0 to 3.1 with HoeffdingTree: AUC improves as history data volume increases.

5 CONCLUSION

The paper proposes a multi-sourced approach to the defect proneness prediction. The approach performs a classification process based on the adoption of a machine learning classifier (different ML algorithms are tested) and a proposed features model (using a mixed set of product and process metrics as features). The approach is tested on a real data set composed of two software systems of totally 183 releases. The obtained results show that the proposed features have an effective defect proneness prediction ability and that such ability is deeply influenced by the project maturity. As future work, the extension effort is twofold: *improve the set of metrics considered as features* — social network metrics impact will be evaluated by considering degree, betweenness, and connectedness of committers in the developer's social network working on source code artifacts; *enforce the empirical validation* — this will be obtained by adding new software projects of different domain and with different structural characteristics.

REFERENCES

Basili, V. R., Briand, L. C., and Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761.

Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. T. (2011). Don't touch my code!: examining the effects of ownership on software quality. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 4–14. ACM.

Boucher, A. and Badri, M. (2016). Using software metrics thresholds to predict fault-prone classes in object-oriented software. In *2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science Engineering (ACIT-CSII-BCD)*, pages 169–176.

Briand, L. C., Wst, J., Daly, J. W., and Porter, D. V. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245 – 273.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

Fischer, M., Pinzger, M., and Gall, H. (2003). Populating a release history database from version control and bug tracking systems. In *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*, pages 23–. IEEE Computer Society.

Gyimothy, T., Ferenc, R., and Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910.

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA. IEEE Computer Society.

Isong, B., Ifeoma, O., and Mbodila, M. (2016). Supplementing object-oriented software change impact analysis with fault-proneness prediction. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pages 1–8.

Kanmani, S., Uthariaraj, V. R., Sankaranarayanan, V., and Thambidurai, P. (2007). Object-oriented software fault prediction using neural networks. *Inf. Softw. Technol.*, 49(5):483–492.

Kapila, H. and Singh, S. (2013). Article: Analysis of ck metrics to predict software fault-proneness using bayesian inference. *International Journal of Computer Applications*, 74(2):1–4. Full text available.

Kaur, A. and Kaur, I. (2018). An empirical evaluation of classification algorithms for fault prediction in open source projects. *Journal of King Saud University - Computer and Information Sciences*, 30(1):2 – 17.

Kim, S., Whitehead, E. J., and Zhang, Y. (2008). Classifying software changes: Clean or buggy? *IEEE Trans. Software Eng.*, 34(2):181–196.

Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496.

Moser, R., Pedrycz, W., and Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA. ACM.

Nagappan, N., Williams, L., Vouk, M., and Osborne, J. (2005). Early estimation of software quality using in-process testing metrics: A controlled case study. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7.

Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., and Murphy, B. (2010). Change bursts as defect predictors. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 309–318, Washington, DC, USA. IEEE Computer Society.

Olague, H. M., Etzkorn, L. H., Gholston, S., and Quattlebaum, S. (2007). Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on Software Engineering*, 33(6):402–419.

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

Rodriguez, J. D., Perez, A., and Lozano, J. A. (2010). Sensitivity analysis of k-fold cross validation in prediction error estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(3):569–575.

Spinellis, D. (2005). Tool writing: a forgotten art? (software tools). *IEEE Software*, 22(4):9–11.