Defect Prediction on a Legacy Industrial Software: A Case Study on Software with Few Defects

Yavuz Koroglu
yavuz.koroglu@boun.edu.tr

Alper Sen
alper.sen@boun.edu.tr

Department of Computer Engineering
Bogazici University, Istanbul, Turkey

Doruk Kutluay
dkutluay@netas.com.tr

Akin Bayraktar
akinb@netas.com.tr

Yalcin Tosun
ytosun@netas.com.tr

Murat Cinar
muratc@netas.com.tr

Hasan Kaya
hasank@netas.com.tr

Netas Telecommunications
Istanbul, Turkey

ABSTRACT

Context: Building defect prediction models for software projects is helpful for reducing the effort in locating defects. In this paper, we share our experiences in building a defect prediction model for a large industrial software project. We extract product and process metrics to build models and show that we can build an accurate defect prediction model even when 4% of the software is defective.

Objective: Our goal in this project is to integrate a defect predictor into the continuous integration (CI) cycle of a large software project and decrease the effort in testing.

Method: We present our approach in the form of an experience report. Specifically, we collected data from seven older versions of the software project and used additional features to predict defects of current versions. We compared several classification techniques including Naive Bayes, Decision Trees, and Random Forest and resampled our training data to present the company with the most accurate defect predictor.

Results: Our results indicate that we can focus testing efforts by guiding the test team to only 8% of the software where 53% of actual defects can be found. Our model has 90% accuracy.

Conclusion: We produce a defect prediction model with high accuracy for a software with defect rate of 4%. Our model uses Random Forest, that which we show has more predictive power than Naive Bayes, Logistic Regression and Decision Trees in our case.

CESI '16 Austin, Texas USA

DOI: http://dx.doi.org/10.1145/2896839.2896843

CCS Concepts

•**Software and its engineering** → **Maintaining software;** •**Computing methodologies** → *Supervised learning by classification;* Classification and regression trees;

Keywords

Defect Prediction; Experience Report; Process Metrics; Feature Selection; Random Forest

1. INTRODUCTION

The complexity of software systems is increasing with the increasing demands of the industry. Hence, systematic analysis of the developed software before it reaches the customer becomes a major challenge. In fact, most software is shipped with defects in them. Testing is the most commonly used approach for detecting defects, however the scalability of traditional testing approaches poses a problem with the increasing size of the software. Defect prediction models have been successfully used to direct testing efforts to probable causes of defects in the software. These models employ software metrics such as product, process, as well as people.

We constructed a defect prediction model for a legacy software for one of the leading systems integration companies in Turkey, called Netas (previously Nortel Netas). The company employs test teams, however, due to limited time and resources, test teams can not sufficiently test the software. The company has an R&D facility since 1973, and the legacy software used in this study is under development since 2012. The legacy software is a VOIP telecommunications software and contains around 35K Java classes. The company announces monthly releases and is planning to produce weekly patches to the software while they move to a continuous integration system. Therefore, there is not enough time to sufficiently test the whole project until the next patch.

We collected software product and process metrics according to a previous study which investigates the inclusion of process metrics on top of product metrics for defect prediction. We also added four process metrics (*Average Bug Criticality*, *Average Bug Fixes*, *Previous Bug Criticality* and

Previous Bug Fixes) on top of the already used process metrics. The relationship with older bug fixes and bug criticality with defectiveness is intuitive since these metrics are directly related to the previous defects of the Java class. We collected these metrics for both the previous version and the average of all older versions. We show that the additional metrics increase the predictive ability of our model.

We share the following experiences in this paper. We present our approach to mine metrics from the software and show that they improve the predictive ability of our model. Then, we compare different machine learning algorithms and adjust their parameters to produce the best predictor for the software. Our model guides the testing effort to only 8% of the project where 53% of the defects reside.

Defect prediction is applied in wide variety of software and is not new in Turkey. Tosun et al. [20] have developed a defect prediction model based for a Turkish telecommunications company as we did. They use Naive Bayes, whereas we compare Naive Bayes, Decision Tree, Logistic Regression and Random Forests. Random Forests are previously used directly in defect prediction, Chug and Dhall [5] utilize Random Forests for defect prediction and report comparatively better results for Random Forests over other models. The metrics we used mostly come from studies of Madeyski and Jureczko [13] and D'Ambros et al [7]. Malhotra presents results of defect prediction for software with few defects [14]. Our work can be viewed as a defect prediction study of a software with few defects. We utilize both process and product metrics and compare different models to come up with the best performing defect prediction model for the legacy software.

We explicitly define our goals aligned with the objectives of the company in Section 2. We describe the main challenges and our approach in Section 3. We discuss the results in Section 4. We explain the threats to validity in Section 5 and share our best practices and important mistakes in Section 6. We describe similar work in Section 7. We conclude with future suggestions and a summary of our work in Section 8.

2. GOALS

Our main objective in this project was to integrate a defect predictor into the continuous integration (CI) process of a large software project and decrease the testing effort. One of the requirements for our project was that the test team wanted to test only a small portion, that is 10%, of all Java classes, and still be able to find a high percentage of the defects. This will allow the company to make releases faster.

To guide the test team, our model should be able to classify Java classes as either *Defective* or *Non-Defective*. Predictive ability of such a classifier is measured from its *Confusion Matrix*, where a confusion matrix is a table that is used to describe the performance of a classifier on a set of test data for which the true values are known. Such a confusion matrix is given in Table 1 where 1 denotes defective and 0 denotes non-defective. Similarly, t_p, f_p, t_n and f_n denote number of true positives, false positives, true negatives, and false negatives, respectively.

The company challenged us to label 10% of all classes in the legacy project as *Defective*. Therefore, the *Positive Prevalence* $((t_p + f_p)/(t_p + f_p + t_n + f_n))$ of our predictor should be less than or equal to 10%.

Our second goal is to make test team find most of the actual defects by inspecting this 10%. Therefore, the *Recall* $(t_p/(t_p + f_n))$ of our predictor should be as close to 1 as possible. To achieve this, we should minimize *false negatives* (f_n).

Table 1: Confusion Matrix for a Defect Predictor

		predicted	
		0	1
actual	0	t_n	f_p
	1	f_n	t_p

We planned our work in three main phases:

- In the first phase, we collected metrics at class level for all versions. We obtained the ground truth on defective classes for each old version through the issue tracking system.
- In the second phase, we used a machine learning technique to build a predictive model.
- In the third phase, we used the predictive model to classify defects in a future release.

First, we analyze the predictive ability of our model on the training set and predict the previous version, using the data coming from all versions before the previous version. Then, we use the best performing learning technique to learn from all previous versions and predict the current release. We use the results to guide the testing effort.

3. METHODOLOGY

3.1 Metrics

In defect prediction studies, several product and process metrics are used. We give a brief description of these metrics as well as the additional process metrics that we introduce.

3.1.1 Product Metrics

Product metrics are static code attributes that are specific to a given version of the software. Several product metrics have been explored in the literature for defect prediction [13]. We used CKJM Extended [6, 17] to extract the product metrics from the source code. The company provided us with the product metrics given in Table 2 as the source code of the project was not available to us. The detailed description of these product metrics can be found in [15, 17].

3.1.2 Process Metrics

Process metrics are attributes related to the change of the program unit throughout previous versions of the software (a program unit is a Java class in our case). Madeyski and Jureczko define process metrics which are known to improve predictive ability when used with product metrics [13]. We collected the process metrics shown in Table 3. Although we were not able to correctly measure NML (i.e. number of modified lines of a class between two releases) due to unavailability of the required knowledge in JIRA and ClearCase databases, we calculated the difference in LOC as a similar measure. There is also a fourth metric NR (number of revisions) used in [13], however this was not readily available from the version control tool that the company used.

Table 2: Collected Product Metrics

#	Metric	Description
1	WMC	Weighted Method Count
2	DIT	Depth of Inheritance Tree
3	NOC	Number of Children
4	CBO	Coupling Between Objects
5	RFC	Response for Class
6	LCOM	Lack of Cohesion in Methods
7	Ca	Afferent Couplings
8	Ce	Efferent Couplings
9	NPM	Number of Public Methods
10	LCOM3	Lack of Cohesion in Methods
11	LOC	Lines of Code
12	DAM	Data Access Metric
13	MOA	Measure of Aggregation
14	MFA	Measure of Functional Abstraction
15	CAM	Cohesion Among Methods of Class
16	IC	Inheritance Coupling
17	CBM	Coupling Between Methods
18	AMC	Average Method Complexity

Table 3: Known Process Metrics

#	Metric	Description
1	NDPV	Number of Defects in the Previous Version
2	NML	Number of Modified Lines
3	NDC	Number of Distinct Commiters

The company uses JIRA [11] as an issue tracking tool and ClearCase [2] as a version control utility. From JIRA and ClearCase data, it is possible to track modifications to Java classes at each version using a metric extraction script. We track the version history of a class using class and package names.

Due to moving to a continuous integration platform, the company decided to change around 20% of the packages where Java classes reside. This created a challenge for correctly gathering process metrics, since the history of some classes can not be matched with the classes and therefore it is missing from our data set. The company provided partial information for some of these changes. Although we incorporated the refactoring information into our metric extraction script, we did not eliminate all problems regarding class refactorization.

3.1.3 Additional Process Metrics

Since the legacy software is quite mature with only 4% of the classes as defective, we needed to increase the model accuracy by introducing explanatory features. We introduced four additional process metrics for this purpose in our study. These metrics are about the criticality of bugs and the number of bug fixes for a given class. Another motivation for introducing these metrics is that if the Java class contained a high number of fixed defects with high criticality, then the Java class should have a high defect density. Also, we believe that as developers fix a Java class, they may introduce new defects, which increases defectiveness of the future releases. We collect these additional metrics from JIRA and ClearCase. We show usefulness of the additional metrics by applying several feature selection methods and comparing prediction models with and without these metrics in Section 4.3.

Table 4: Additional Process Metrics

#	Metric	Description
1	PBC	Previous Bug Criticality(1-5)
2	ABC	Average Bug Criticality(1-5)
3	PBF	Previous Bug Fixes
4	ABF	Average Bug Fixes

Table 4 shows these process metrics. In Table 4, first and second rows refer to the criticality of the defect from 1 (non-critical) to 5 (most critical). Each JIRA issue contains such a field denoting the criticality of the issue. Third and fourth rows can be mined by counting the resolved issues of previous versions of the Java class. *Average* means that we take the average number of all previous version data and *Previous* means we take data only from the last version prior to the current.

3.2 Classifiers

After collecting the data, we require a classifier to learn the previous versions and predict the defects in upcoming release. There are several choices available. We know that Naive Bayes, Logistic Regression, J48 Decision Tree and Random Forest are used in defect prediction [16]. We also used voting on two best classifiers to boost predictive ability. We trained each model with different hyperparameters and compared the *Receiver Operational Characteristics* (ROC) of the classifiers as shown in Figure 1. A ROC curve is a graphical plot that is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. We discuss details of our comparison in Section 4.1.

As we discussed before, the software project contains very few defects. Therefore our training set is highly imbalanced. We followed the results of previous work regarding learning highly imbalanced data with Random Forests and used *Synthetic Minority Over-sampling TEchnique* (SMOTE) [3] to balance our training set as suggested in [4] and in [10]. SMOTE oversamples the instances belonging to minority classes in the training set in order to make the class priors roughly equal to each other (i.e. make the training set contain roughly equal number of instances for each class). To make our classifier sensitive enough to defective units, we oversampled the defective instances so that we have 20x more defectives in our training set. After oversampling, class priors of our training set becomes roughly equal.

3.3 Data

The company provided us with data from seven previous releases of the software. They also provided the product metrics of the last release. We used the JIRA entries of older versions of the class to collect the process metrics since process metrics are related to the history of the class. We share the general information of our data set in Table 5. The data set does not contain the number of defects of the last version, since this information is not available to us yet.

To be able to measure the predictive ability of our classifier, we predict defects of version 11.2 and compare with the actual defects found in version 11.2. Our training set to predict defects in version 11.2 contains 144,111 entries (an entry for each class of each version) from all 10.x versions

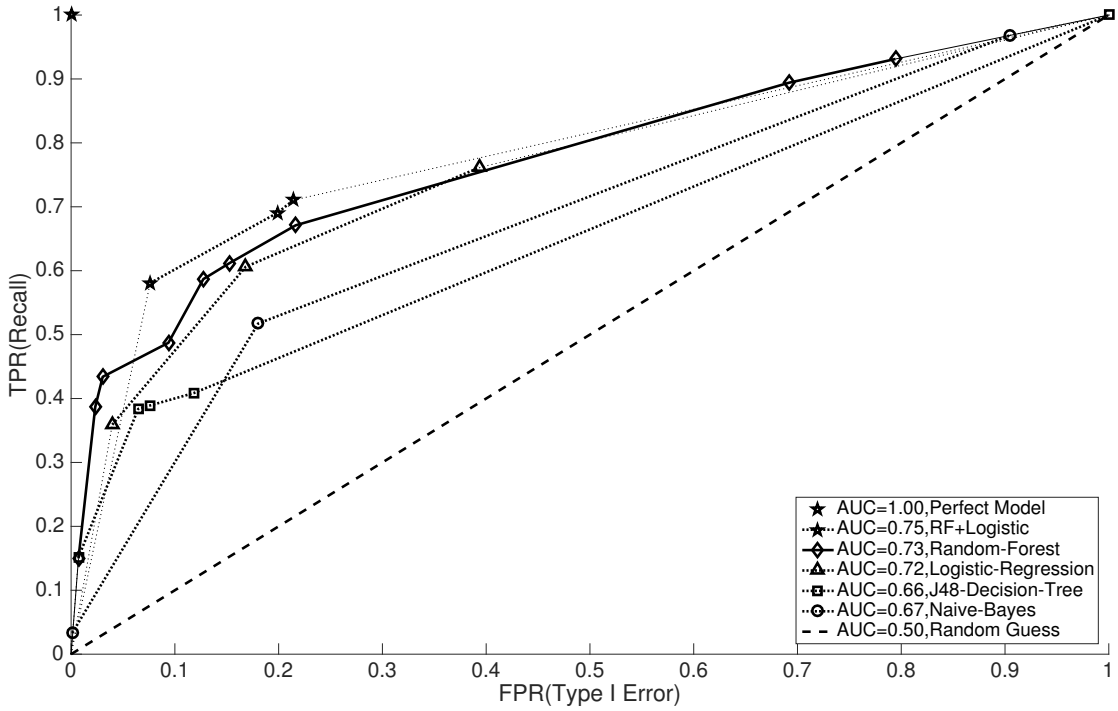

Figure 1: Receiver Operational Characteristics Curve.

Table 5: Data Set

Version	# Classes	# Defective	% Defective
10.0	31758	1584	5%
10.1	32600	1725	5%
10.2	33332	1273	4%
10.3	34702	1920	5%
10.4	37554	1005	3%
11.2	37988	1295	3%
12.0	36089	Not Available	Not Available

where 5923 of these entries are defective. Each entry contains 18 product and 7 process metrics and a boolean value denoting defective/non-defective, in total 26 attributes.

4. RESULTS

We used WEKA tool [9] to train classifiers, apply feature selection techniques and produce prediction reports on defective classes and confusion matrices. WEKA is a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to a dataset or called from a Java code. We applied different classification techniques directly.

4.1 Comparison of Classifiers

Predictive ability of multiple classifiers can be compared by measuring the *Area Under Curve* (AUC) of *Receiver Operational Characteristics* (ROC) of each classifier [1]. ROC is a curve on two dimensional space where x axis is *False Positive Rate* (i.e. $f_p/(f_p + t_n)$) and y axis is the *True Positive Rate* (i.e. $t_p/(t_p + f_n)$). For each hyper-parameter setting, we measure *True Positive Rate* and *False Positive Rate* of the classifier. These measurements form points on the two dimensional space. When we plot all measured points for the classifier, we get the ROC curve. Then we calculate the area under ROC curve (AUC) which is a measure of performance of the classifier. A perfect classifier's *False Positive Rate* is zero and *True Positive Rate* is one, so the perfect classifier is a point on ROC curve. A completely random classifier would have equal true and false positive rates, therefore a random classifier is a diagonal line on the ROC curve. We expect any good classifier to be above the random curve and close to the perfect point (0.0, 1.0). For different hyperparameters, we share our measurements in Figure 1.

Table 6: Confusion Matrix of Random Forest on Version 11.2

	0	1
0	34242	2451
1	612	683

An ensemble technique, voting between Random Forest and Logistic Regression techniques produces the best AUC value. However, since the ensemble model requires too much time to learn and the company has to consider time limitations, we discarded voting. We chose to use Random Forest since it is faster and produces the second best AUC.

4.2 Overall Results

We present the confusion matrix produced by the Random

Table 7: Performance Measurements of Random Forest

Measure	Value
Recall (R)	0.527
Precision (P)	0.218
Accuracy (A)	0.919
Positive Prevalence	0.087

Forest on predictions of version 11.2 in Table 6. We present *Recall*, *Precision* and *Accuracy* as a measure of our model's predictive ability. We use the *Positive Prevalence* to show that we are in the boundaries of our goal. We present our measurements in Table 7. We present the definitions of these measures in Equations 1, 2, 3 and 4.

$$\text{Recall}(R) = \frac{t_p}{t_p + f_n} \quad (1)$$

$$\text{Precision}(P) = \frac{t_p}{t_p + f_p} \quad (2)$$

$$\text{Accuracy}(A) = \frac{t_p + t_n}{t_p + f_p + t_n + f_n} \quad (3)$$

$$\text{Positive Prevalence} = \frac{t_p + f_p}{t_p + f_p + t_n + f_n} \quad (4)$$

We state the following conclusions from the results:

- From *Positive Prevalence*, we conclude that we can guide the test team to inspect only 8.7% of the project classes.
- From *Precision*, we state that the test team is expected to find defects in 21.8% of the predicted classes.
- From *Recall*, we state that the 8.7% of the project contains 52.7% of the actual defects.
- As a last note, our model has a high overall accuracy (90%).

4.3 Usefulness of Additional Metrics

In this section, we discuss usefulness of the four additional metrics we use in Table 4, namely ABC, ABF, PBC and PBF.

4.3.1 Feature Selection Results

We used feature selection techniques to select the most useful subset of all metrics. We used the techniques called *GreedyStepwiseSearch* and *BestFirstSearch* in both forward and backward directions. We used *CfsSubsetEval* as attribute evaluator. Table 8 shows the occurrence of ABC, ABF, PBC and PBF metrics after each subset selection. We can see that ABC and ABF exist in all most useful metric subsets.

We ranked the attributes according to their information gain using *GainRatioAttributeEval* evaluator. We present the first 10 metrics after evaluation in Table 9. We can see that PBF and PBC are ranked high. In conjunction with the results given in Table 8, we believe that the contribution of all new metrics is significant to our model.

4.3.2 Field Results

We trained our best classifier, Random Forest, without our additional metrics to predict defects in version 11.2. We present the resulting confusion matrix in Table 10 and the performance measures in Table 11. Without the additional metrics we get lower *Precision*, *Recall* and *Accuracy* values and higher *Positive Prevalence*. With additional metrics, we get 32.1% higher *Precision*, 18.7% higher *Recall* and 1.5% higher *Accuracy* with 3.3% lower *Positive Prevalence* over the results without the additional metrics. We calculated these results using 5, for example we calculated the increase in *Precision* as $(0.218 - 0.165)/0.165 = 0.321$. Therefore, we conclude that the additional metrics impact defectiveness and are useful for finding defects in the legacy software.

$$\text{Relative Increase} = \frac{\text{Current} - \text{Previous}}{\text{Previous}} \quad (5)$$

5. THREATS TO VALIDITY

The correctness of the collected product metrics is an issue. Since CKJM Extended calculates the LOC from the bytecode, it may not reflect the correct number of lines of source code. We assume there is a certain correlation between bytecode LOC and source LOC. Furthermore, even if the bytecode LOC is different from the source LOC, we believe that the binary LOC is still a reasonable metric for defect prediction.

The correctness of the collected process metrics is also an issue. Since we are tracking classes by class and package names, in case of unhandled refactorings we assume that the old class file is deleted and a new class is created from scratch. This directly affects all process metrics collected for that class. 16,094 of 144,111 Java classes in our data set has no version history and therefore have all zeros as their process attributes. Previous work suggests that if we had a way to correctly track the refactored classes, we could increase the *recall* and *accuracy* of our model [19].

We claim that we can guide the test effort to 8% of the project. However, this claim is not entirely correct. If we measure the total lines of code of the classes we predict as defective, it may be well above or below 8% of total lines of code of the project. Still, we believe that class-level granularity approximately reflects the decrease in the test team effort.

We only used the legacy software of the company to train and use our defect prediction model. Without conducting experiments on different software, any result we discuss in this paper may not be generalized.

6. LESSONS LEARNED

For companies that collect version histories and other data for their projects, it is safe to conclude that the collected data needs refinement before it can be used in defect prediction.

The company we worked with uses SonarQube [18] to collect product metrics. However, several complex SQL queries are required to run on different versions of SonarQube to gather all the product metrics used in this work. Therefore, we suggested to use CKJM Extended as an alternative.

We learned that it is possible to track changes in classes by just inspecting JIRA and ClearCase data. We were able to incorporate the data and extra information given by the

Table 8: Occurrence of the Additional Metrics after Feature Selection

Selection Method	Search Direction	ABC	ABF	PBC	PBF
BestFirst	Forward	√	√		
BestFirst	Backward	√	√		
GreedyStepwiseSearch	Forward	√	√		
GreedyStepwiseSearch	Backward	√	√		

Table 9: First Ten Metrics According to Their Individual Information Gain

Rank	Metric
1	NDC
2	NDPV
3	PBF
4	NML
5	PBC
6	ABF
7	ABC
8	DIT
9	Ca
10	MOA

Table 10: Confusion Matrix of Random Forest without Additional Metrics

	0	1
0	33793	2900
1	720	575

company to our model. This extra information allowed us to easily obtain additional metrics (ABF, PBF, ABC, PBC). We believe that it is essential to use any knowledge that can increase predictive ability of trained models like these additional metrics.

When faced with a highly imbalanced data, we learned that oversampling allows more flexible models than using a cost function.

The company also provided us with the weekly patch data on top of the monthly release data. However, the new data did not result in an improvement over the current model. Therefore we believe that either patch data is irrelevant to our dataset or the predictive ability is not increasing because of the irregularities in the data. Hence we did not use them in our experiments.

7. RELATED WORK

Moeyersoms et al. [16] use a rule extraction algorithm called ALPA to produce comprehensible defect predictors. They use Random Forests to generate trees and use the trees for the rule extraction whereas we directly use a Random Forest as the defect prediction model. They claim ALPA model is more accurate than the previous models which work directly on the data. Our model has roughly the same *Accuracy* and *Recall* values with the best measured values of ALPA model. We are not able to compare *Precision* since they did not share that information.

Table 11: Performance Measurements of Random Forest without Additional Metrics

Measure	Value
Recall (R)	0.444
Precision (P)	0.165
Accuracy (A)	0.905
Positive Prevalence	0.09

Malhotra presents AUC values of many different defect prediction models in a systematic review [14]. For cases where actual defect rates are below 5%, our model has a better AUC value than the reported methods.

Ghotra et al. [8] calculate AUC values of different classification techniques on many projects. In comparison with their work, the AUC value of our defect prediction model (0.73) stands better than J48 Decision Tree and comparable to Naive Bayes and ensemble approaches involving Rotation Forest.

Tosun et al. [20] use Naive Bayes as defect prediction model in a case study of Turkish communications industry. They use microsampling instead of oversampling. Microsampling reduces the amount of non-defective instances so that it matches the amount of defective instances. In our case, we discard microsampling since it produces a very small training set.

Most of the metrics we use come from a study on the process metrics by Madeyski and Jureczko [13]. They used Stepwise Linear Regression to train their prediction model. We add more process metrics on top of the proposed metrics and we use Random Forest, since we are more interested in the predictive ability of our model than the usefulness of additional metrics.

Among the metrics D'Ambros et al. [7] use in their extensive study of bug prediction approaches, 10 of them were common to our metrics (NML, DIT, WMC, RFC, CBO, LCOM, Ca, Ce, NPM, PBF). They also use one of the additional metrics we use, *Previous Bug Fixes* (PBF). They report that using PBF performs better than the other product and process metrics.

Regression Testing [21] is used to eliminate redundant test cases and reduce test effort by decreasing the test suite size. With our prediction model, the test team is guided to write more tests for dangerous classes and fewer tests for the safe ones. Therefore, we argue that the resulting test suite will have less redundancy and decrease regression testing effort [12].

8. CONCLUSIONS

Our defect prediction model enables the reduction of the overall test effort. By inspecting only 8% of the software project, we correctly predict 53% of all known defects in the software. Our additional metrics increase both the *Precision* and the *Recall* of our model by 32% and 18% over the results without additional metrics, respectively. We show that by comparing AUC values, Random Forest produces

defect prediction models with better predictive ability than other learning techniques such as Naive Bayes, Logistic Regression, and Decision Trees. Our prediction model is incorporated into the continuous integration pipeline and is starting to be used on a new version of the software. We believe that the new results will increase confidence in our defect prediction model. In the future we plan to implement our model as an *online* algorithm, which learns with each release. Also, we plan to replicate our study on different companies and projects, to increase external validity.

9. REFERENCES

[1] E. Alpaydin. *Introduction to Machine Learning*. The MIT Press, 3rd edition, 2014.

[2] ClearCase, http://www-03.ibm.com/software/products/en/clearcase.

[3] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, June 2002.

[4] C. Chen, A. Liaw, and L. Breiman. Using Random Forest to Learn Imbalanced Data. Technical report, Department of Statistics, University of Berkeley, 2004.

[5] A. Chug and S. Dhall. Software defect prediction using supervised learning algorithm and unsupervised learning algorithm. In *Confluence 2013: The Next Generation Information Technology Summit (4th International Conference)*, pages 173–179. IET, 2013.

[6] CKJM Extended, http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/.

[7] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, Aug. 2012.

[8] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, 2015.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[10] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.*, 38(6):1276–1304, Nov. 2012.

[11] JIRA, https://www.atlassian.com/software/jira.

[12] Y. Kastro and A. B. Bener. A defect prediction method for software versioning. *Software Quality Journal*, 16(4):543–562, Dec. 2008.

[13] L. Madeyski and M. Jureczko. Which process metrics can significantly improve defect prediction models? an empirical study. *Software Quality Journal*, 23(3):393–422, Sept. 2015.

[14] R. Malhotra. A systematic review of machine learning techniques for software fault prediction. *Appl. Soft Comput.*, 27(C):504–518, Feb. 2015.

[15] Description of Product Metrics, http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html.

[16] J. Moeyersoms, E. J. de Fortuny, K. Dejaeger, B. Baesens, and D. Martens. Comprehensible software fault and effort prediction: A data mining approach. *Journal of Systems and Software*, 100:80–90, 2015.

[17] D. Spinellis. Tool writing: A forgotten art? *IEEE Softw.*, 22(4):9–11, July 2005.

[18] SonarQube, http://www.sonarqube.org.

[19] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, 2015.

[20] A. Tosun, A. Bener, B. Turhan, and T. Menzies. Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. *Inf. Softw. Technol.*, 52(11):1242–1257, Nov. 2010.

[21] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Mar. 2012.