Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1357–1363 DOI: 10.15439/2017F536 ISSN 2300-5963 ACSIS, Vol. 11

A case study on machine learning model for code review expert system in software engineering

Michał Madera
Rzeszów University of Technology
al. Powstańców Warszawy 12,
35-959 Rzeszów
Poland
Email: michalmadera@gmail.com

Rafał Tomoń
SoftSystem Sp. z o.o.
ul. Leszka Czarnego 6a,
35-615 Rzeszów
Poland
Email: rtomon@softsystem.pl

***Abstract*—Code review is a key tool for quality assurance in software development. It is intended to find coding mistakes overlooked during development phase and lower risk of bugs in final product. In large and complex projects accurate code review is a challenging task. As code review depends on individual reviewer predisposition there is certain margin of source code changes that is not checked as it should. In this paper we propose machine learning approach for pointing project artifacts that are significantly at risk of failure. Planning and adjusting quality assurance (QA) activities could strongly benefit from accurate estimation of software areas endangered by defects. Extended code review could be directed there. The proposed approach has been evaluated for feasibility on large medical software project. Significant work was done to extract features from heterogeneous production data, leading to good predictive model. Our preliminary research results were considered worthy of implementation in the company where the research has been conducted, thus opening the opportunities for the continuation of the studies.**

I. Introduction

DEFINING QA processes is a challenging task for organizations developing software-intensive systems. QA efforts could strongly benefit from accurate estimation of error prone project areas. Taking into account relative cost of fixing software defects based on time of detection, any improvements in early development stage are worth the effort (Fig. 1). The National Institute of Standards and Technology (NIST) estimates that code fixes performed after release can result in 30 times the cost of fixes performed during the design phase [1]. Additional costs may include a significant loss of productivity and confidence. The NIST report also indicates, that involving programmers in tracking and correcting their own errors, by reviewing code before run time testing improves their programming skills. Curhan states that “some types of defects have a much higher costs to fix due to the customer impact and the time needed to fix them or the wide distribution of the software in which they are embedded” [2]. Adam Kolawa [3] defines error as a human mistake and a defect as a fault, bug, inaccuracy, or lack of expected functionality in a project artifact. Broad definition of defects, thus includes problems such as contradicting requirements, design oversights or coding bugs. With proper processes for requirements and design review in place, when building prediction model we are focusing on coding problems only. In fact, every software moved to production contains defects, although many are not detected yet. Thus, undeniable increase of corrections costs for subsequent project stages lead to conclusion that major effort should be put in the earliest possible phases of software production process. We assume that best place to focus is code review stage, a place in the process, when we have a chance to eliminate the problems at its genesis (Fig. 2) McIntosh et. al. [4] summarized their case study with statement: “Components with higher review coverage tend to have fewer post-release defects”. Their analysis also indicates that “Although review coverage is negatively associated with software quality in our models, several defect-prone components have high coverage rates, suggesting that other properties of the code review process are at play.” In our situation, company with full code review coverage, still face relatively large number of defects reported.

We put the thesis, that with predicting code changes failures we can direct more focus on endangered areas, with additional code review, and have potential defects corrected before testing phase. That would significantly improve final software quality, with lower operational costs. Important part of this research is feature engineering that allows building reliable problem prediction model. There is also proposed approach for software development process with defect prevention mechanism, improving QA effectiveness in large and complex software projects. Particular attention is paid to having the research applicable in real life scenarios. Our preliminary research results were considered worthy of implementation in the company. The authors’ contribution to the work is feature set development, data mining from variety of sources, feature selection and building reliable prediction model. We explicitly define our goals aligned the objectives of company in Section 2. We explain how the results will be evaluated in Section 3. Data acquisition caveats and the data set are presented in Section 4. We present main challenges and taken approach to getting results in Section 5. We conclude with summary of our work and future suggestions in Section 6, followed by Appendix with list of all attributes acquired for this research.

This work was supported by SoftSystem Sp. z o.o.

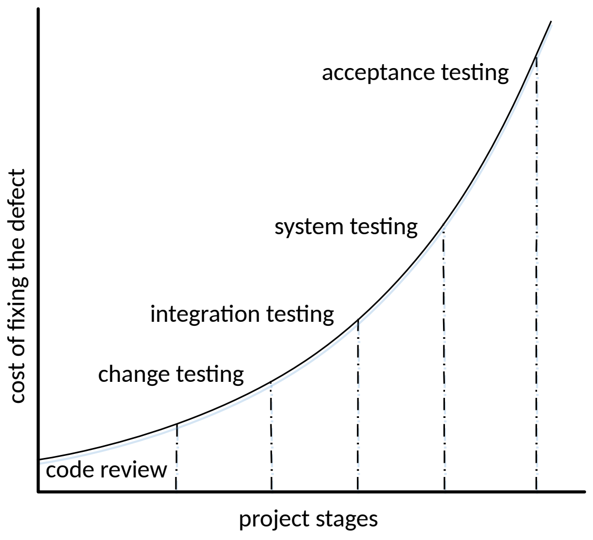

Fig 1. Relative cost to fix, based on time of detection

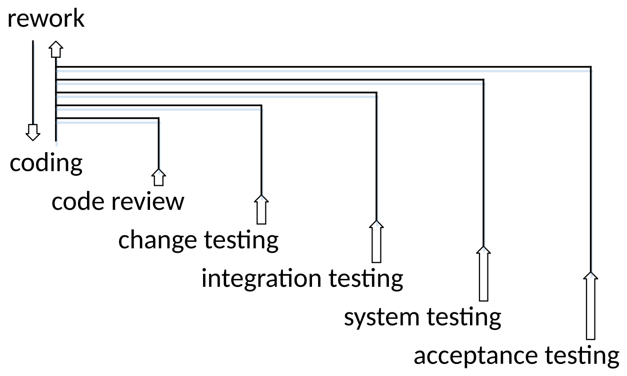

Fig 2. Software development process stages with flow of coding tasks rejections (reworks)

II. Goals

The main objective in this work was to build problem predictor model that could be integrated into process of a large software project to support decisions and improve quality of final products. The paper is a case study on building *Rework* prediction system for one of the leading companies developing software for hospitals and medical laboratories around the world. Company established QA department over 15 years ago and employs highly qualified testing team working according to best industry standards. Along with manual testing there is also automated testing team that is continuously monitoring product quality. The company has been certified to the ISO 9001:2008 and ISO 13485:2012 standards [5]. With the requirements for a quality management system, specific to the medical devices industry, the company is expected to constantly improve quality management processes. A good example of such actions is implementing static code analysis process, which increased source code overall quality, but it has not had significant influence on number of defects in general. Company introduces changes in processes on different organizational levels to improve quality, and our research is part of these endeavors.

Number of source code change reworks is constant for last three years (Fig. 3). This period will be a base for building a prediction model. Rework stands for "change implemented by programmer that was rejected, qualified for correction either by code reviewer or testing team". Software change can be rejected for multiple reasons and our analysis will focus on following categories:

- Source code review failed
    - Source changes rejected by programmer
    - Static code analysis problems detected
- Functional testing failed
    - Manual testing (change, integration, etc.)
    - Automated testing (acceptance, regression)

The expected problem detection moment is when the programmer completed the work. Ideally, the programmer who is implementing the change, reviews the code and corrects bugs that were introduced, before passing the finished work for code review by other programmer and testing. Also acceptable situation is when code reviewer (technical team leader) is able to track down the problems and move the implementation back for corrections. Our goal is to support this flow of events. The complete change called "an issue" consists of functional requirements, design documents and files that were modified and submitted to Apache Subversion (SVN), a software versioning and revision control system. In this work we consider reworks as individual files, that were submitted to SVN after rejection of the whole change. The assumption was made, that these files are "reworked" and, they require additional changes or corrections. That is not true for every instance but acceptable for purpose of building the model in this research.

The company requested to have reliable information which source code changes require extensive code review. Information about records with increased risk should be delivered as soon as programmer completes the work and marks the task as ready for review. To guide development

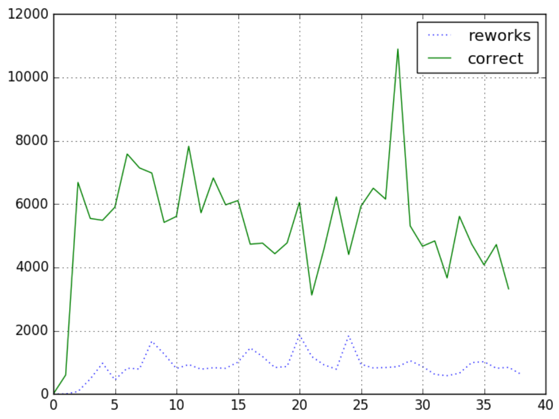

Fig 3. Rejected code changes vs. correct changes, in years 2014-2016, monthly

teams, our model should be able to classify particular file changes either *Correct* or *Rework*.

III. Model evaluation

Performance of the classifier is measured with Confusion Matrix, which is a table describing predictive ability of a classifier on test data set for which the true values are known, as in Table 1. *Rework* denotes positive value and *Correct* denotes negative value. Tp, Fp, Tn, and Fn denote the number of true positives, false positives, true negatives, and false negatives, respectively. Predictive ability of multiple models can be compared by measuring the area under receiver operational characteristics (AUROC) of each classifier [6]. Receiver operating characteristic (ROC) is a curve on two dimensional space where x axis is False Positive Rate (i.e. Fp/(Fp+Fn)) while y axis is the True Positive Rate (i.e. Tp/(Tp+Tn)). We measure True Positive Rate and False Positive Rate of the classifier and the set of attributes. Measurements are taken directly from Weka software. A perfect classifier's False Positive Rate is zero and True Positive Rate is one, thus 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).

Table I.
Confusion matrix for rework predictor

		Predicted	
		Correct (negative)	Rework (positive)
Actual	Correct (negative)	T_n	F_p
	Rework (positive)	F_n	T_p

Results of the classification can be said to be precise if the values are close to the average value of the quantity being measured, while the results can be said to be accurate if the values are close to the true value of the quantity being measured. Precision and accuracy are defined [7] as follows:

$$Precision=\frac{T_p}{T_p+F_p} \tag{1}$$

$$Accuracy=\frac{T_p+T_n}{T_p+F_p+T_n+F_n} \tag{2}$$

Sensitivity is the probability that a model will indicate *Reworks* among those which actually are *Reworks*:

$$Sensitivity=\frac{T_p}{T_p+F_n} \tag{3}$$

Specificity is the fraction of *Correct* records which will be qualified as *Correct*

$$Specificity=\frac{T_n}{T_n+F_p} \tag{4}$$

Sensitivity and specificity are characteristics of the model that does not depend on Correct and Rework proportions. Although in our situation significant classes imbalance has to be taken into consideration as there are only 17% Reworks in data. Thus, important variable in model evaluation is the prevalence of the *Reworks* in question. Prevalence is defined as the percent of instances in the test set that actually are *Reworks*.

$$Positive\,Prevalence=\frac{T_p+F_p}{T_p+F_p+T_n+F_n} \tag{5}$$

Development team would like to get answer to question: what is the chance that a file change classified as a *Rework* truly is a *Rework*? If classified record is in the second row of Table 1, what is the probability of being T_p as compared to F_p? An answer to these questions would be Positive Predictive Value (PPV) and Negative Predictive Value (NPV). Both PPV and NPV are influenced by the positive prevalence of *Rework* instances in the test set. If we test in a high prevalence setting, it is more likely that instances qualified as *Rework* truly are *Reworks* than if the testing is performed in a set with low prevalence.

$$Positive\,Predivtive\,Value=\frac{T_p}{T_p+F_p} \tag{6}$$

$$Negative\,Predivtive\,Value=\frac{T_n}{T_n+F_n} \tag{7}$$

IV. Data

The data set used to build and test the model consists of files changes registered during development of medical laboratory software, between years 2014-2016. The data set contains 237128 observations and a large number of explanatory variables (20 nominal, 6 ordinal and 51 numeric) involved in assessing file change values. Data were collected from many data sources (project management database, issue tracker, requirements library, human resource management database, source control repository, and code metrics software) and consolidated into one data set. Each record is an individual file, changed in context of bigger entity which is a task. For each record (file change) there has been class assigned, appropriately *Correct* or *Rework*.

The combination of data from different systems was possible thanks to the processes that were introduced by the QA department and good integration of these systems. To commit changes to source control (in our case it was SVN), programmer should have created earlier and approved valid task number. SVN was configured in the way that changes with wrong task number were rejected. If a programmer passes valid task number, information about all changed files with change type (A- added, U – updated, D – deleted) and information about author are stored in issue tracker system. This way data about all revisions are stored in system.

The company uses proprietary system for development management called SoftDev.

To query data from the company systems related to development management we used SQL queries. For manipulating the data, we used Java JDMP [8] and Python Pandas [9] libraries. The data was stored in CSV (comma separated value) format. For basic data set, we have created collection of additional mining scripts that queried other systems (source control system (SVN), HR database, requirements library and code metrics software) for contextual data.

In our rework prediction research, large number of features has been collected and grouped into following five categories:

- Employee metrics – metrics containing information about author of file change. All attributes are time aware and are in reference when change was made. We measure for example experience in module(MX) affected by change, experience in sub-module (DMX) as well as experience in file (UEXP).
- Task metrics – set of metrics related to change request.
- Changed file metrics – attributes related to modified file.
- Change quantitative metrics – metrics of file change size.
- Source code metrics – metrics obtained from static code analysis using tool SourceMonitor [10].

All attributes description is available in the appendix.

Medical laboratory system that is subject of this research is developed in Java and .Net programming languages. Client part, Graphic User Interface (GUI) is built with .Net Windows Forms technology. Server part in Java implements logic, database operations, and exposes web-services for GUI. Most of coding tasks require changes in both .Net and Java classes. This limits the number of source code metrics only to those applicable to both technologies. The project is modular with about 4 million Java lines of code and 7 million .Net lines of code. All source code files for the analysis retrieved from source control repository took challenging 85 GB of disk space. This is caused by development running on several branch lines that are separated from main development for months or years. All these development lines were included in the research.

Acquired historical data (2014-2016), with only Java and .Net files has 237128 records, which gives around 300 files committed a day, where 17% was marked as a *Rework*. Learning from historical instances we could predict which changes will be *Reworked* and assign it for more extensive review (review could be done by additional programmers or architects). With 17% detected defects there is certain amount of overlooked problems that will be included in release version, but some of them could have been discovered if critical changes were reviewed more carefully. In defect prediction studies [11],[12], both process and source code metrics are used. We were not able to get satisfactory results with commonly used attributes as well as [13] and [14]. With company software development experts, we worked out long list of attributes that are worth including in prediction model. Apart from attributes commonly used in defect prediction practices, we tried to build other, like these describing employee metrics with experience per module (on different modularization levels), employee history in terms of failures, task complexity related to number of functional requirements or number of people involved in task coding. Final list contains 77 attributes.

V. Results

We used WEKA 3.8.1 [15] software package to build classifiers, select features and produce prediction reports with metrics described in section II. WEKA is an open-source package initiated by University of Waikato, with rich collection of machine learning algorithms for data mining tasks. The algorithms can be either applied directly to a data set or from the Java source code. Due to large data set in our research (223712 records with 77 attributes) we had to use distributed computing when evaluating different classifiers and subsets of attributes.

To reflect real use case for the system, we have trained classifier with 90% of all data and tested with remaining 10%. Assumption is that the model will be built with historical data for certain period, and new instances will be classified with it. Prediction model should be rebuilt every month. Taking into account software development process changes and experience with production cycle specifics we chose 3 years period as input for the model. Results of testing with models built for shorter periods confirmed this decision.

Performance of different classifiers and attribute sets, due to significant imbalance in class distribution, has to be done by measuring the AUROC of each classification [7]. From our experience in this research, the problem of attributes selection was a key aspect to obtain satisfactory results, which was also confirmed by [16], [17], and [18]. With appropriate attributes identified, the better accuracy could be achieved for a smaller sets of attributes with a simple appropriate classifier. We evaluated the following attribute selection algorithms:

- *CorrelationAttributeEval* which evaluates the worth of an attribute by measuring the correlation (Pearson's) between it and the class
- *PrincipalComponents* which performs a principal components analysis and transformation of the data
- ReliefFAttributeEval which evaluates the worth of an attribute by repeatedly sampling an instance and considering the value of the given attribute for the nearest instance of the same and different class.
- *GainRatioAttributeEval* which evaluates the worth of an attribute by measuring the gain ratio with respect to the class
- *InfoGainAttributeEval* which evaluates the worth of an attribute by measuring the information gain with respect to the class

All these attribute selection methods were tested with Weka ranker which ranks attributes by their individual evaluations. Ranker was also used to find best number of attributes, evaluating in range of 15 to 65 attributes with step 6. Combination of 5 classification algorithms, attribute selection methods and number of attributes gave challenging number of 275 cases for grid search. Calculations were done with WEKA distributed computing, using company server resources. The best result has been achieved for Random Forest algorithm with 25 attributes selected with *InfoGainAttributeEval* algorithm.

Performance comparison of selected classifiers is presented in Table II. For chosen list of attributes the Random Forest algorithm provided the highest performance measure by means of AUROC (0.930). The results for other metrics are presented in Tables III-IV.

Table II.
Comparison of different classifiers

Classification algorithm	**Attribute selection algorithm**	**Number of attributes**	**AUROC**
Random Forest	InfoGainAttributeEval	25	0.930
Bayes Net	GainRatioAttributeEval	60	0.816
C4.5	InfoGainAttributeEval	20	0.872
KNN	GainRatioAttributeEval	20	0.829
Naive Bayes	GainRatioAttributeEval	45	0.787

Table III.
Confusion matrix for best prediction model

		Predicted	
		Correct (negative)	Rework (positive)
Actual	Correct (negative)	19755	543
	Rework (positive)	1290	2125

The interpretation of the results for development team is:

If model classifies records as *Correct*, its confidence is 94%. Only 11% of all changed files have to be sent for extended code review, to hit 79% potential problems. Predictive model is able to recognize correctly 62% of all *Reworks*.

We evaluated the classifier using 10 folds cross validation to find out that results are very close to those done in percentage split test. We take this as a confirmation, that prepared model is stable and ready for use in real life scenarios. List of all attributes mined in this research is available in Appendix, with best 25 preselected attributes, marked with a star (*).

Table IV.
Beast prediction model performance metrics

Metric	**Value**
Precision	97.12 %
Accuracy	92.27 %
Sensitivity	62.23 %
Specificity	97.32 %
AUROC	0.930
Positive Prevalence	11.25 %
Positive Predictive Value	79.64 %
Negative Predictive Value	93.87 %

VI. Conclusions

Our *Rework* prediction model will support QA activities with effective estimation of software areas that are at risk by mistakes introduced during source code changes. System will direct extended code review to these places. The proposed approach has been evaluated for feasibility on large medical software project and research results were considered worthy of implementation in the company where the research has been conducted. With very high precision (97.12 %) and accuracy (92.27 %), company should expect visible effects after implementation of the system build upon our research results. We show that by comparing AUROC values, Random Forest provides *Rework* prediction models with better predictive ability then other algorithms like Bayes Net, C4.5, KNN, and Naive Bayes. We believe that sophisticated mechanisms developed to collect the data for this research will be a base for subsequent analysis, and knowledge retrieved will support project management. Subjective medical software project is developed in multiple remote offices in different locations around the world, making the data set even more challenging and interesting from analytical point of view.

Next steps would include incorporating the prediction model into company regular operations and follow up on mechanisms to measure effectiveness of the implementation.

Appendix

Attributes collected from project management database:

ISST	*Problem* category with possible values: Defect, Deficiency, Enhancement, Performance, Refactoring, Coding Standard, Demo, Custom scripts, Test Case, External – 3rd party
ISSS	Severity of issue with possible values: Non Critical, Critical, Risk to Health
ISSP	Priority of issue. Available values: 0-5 (Low - Urgent)
HLE	Task coding time high level estimation in hours (*)
OFF	Employee office name

Attributes collected from issue tracker system:

CR	Task number. Generally task is created from issue and is assigned to programmer.

IMPBY	Person who marked task as 'Implemented' (done). After this action task is passed to testing team.
IMPD	Date when person marked task as 'Implemented'
SOLBY	Person who created solution for task. (Generally, it is more experienced person like architect or team leader)
SOLT	This is set of 20 predefined values describing type of solution.
ECH	Estimated hours for coding based on all details from task (*)
ACH	Actual hours spent on coding. (*)
NCMR	Number of commits for task (*)
NFCR	Number of file changed for task (*)
PINCR	Number of users who committed changes within task
NAM	Number of affected modules (*)
NADM	Number of affected "dipper modules" (*)
NCSR	Number of .net files changed within task (*)
NJVR	Number of java files changed within task (*)
ASM	Number of affected files in the same module (*)
ASDM	Number of affected files in the same "dipper module" (*)
AOM	Number of affected files in other modules (*)
AODM	Number of affected files in "dipper modules" (*)

Data collected from issue tracker system on 'file change level':

CBY	Person who committed file change to source control system
REV	Source control revision related to change
OPT	File change operation: A – addition, D – deletion, U – update
PAT	Absolute path to modified file in source control tree
RPAT	Relative path to modified file
BRA	Source control branch name
PROJ	Project name of modified file
MOD	Module name of modified file
DMOD	Very big modules were divided it into small pieces called "dipper module"
LAN	Coding language of modified file
RWRK	This information stating if particular file change was good or was not. If at least one file was marked as rework, then related task was also marked as rework
URE	This is information on how many reworks has person who committed particular change in his/her history. Attribute was calculated from the whole user history till commit date (*)
URYB	This is information on how many reworks has person who committed particular change over the last year (*)
NOFF	Number of commits of file within task (*)
PIIF	Number of people involved in file within task

Attributes collected from source control (svn):

MCEXP	Sum of all modifications on file made by user who committed change till commit date
TC	Sum of all modifications on file made by all users till commit date. This attribute may be treated as file age measures in changes.
CGTC	File age categorized by expert into 8 categories
UEXP	Contribution of committed user in file till commit date expressed in percentage
CGUX	User Contribution categorized by expert into 6 categories
NFUX	For 63 records we cannot establish user experience
MMX	Sum of all modifications on module made by user who committed change till commit date (*)
DMMX	Sum of all modifications on "dipper module" made by user who committed change till commit date (*)
TMX	Sum of all modifications on module made by all users till commit date. This attribute may be treated as module age measures in changes (*)
TDX	Sum of all modifications on "dipper module" made by all users till commit date. This attribute may be treated as module age measures in changes (*)
MX	Contribution of committed user to module till commit date expressed in percentages (*)
DMX	Contribution of committed user to "dipper module" till commit date expressed in percentage (*)
CMX	User Contribution categorized by expert into 8 categories
CDMX	Same as CMX but on "dipper module" level
DIFI	Number of line insertions on a file in last commit
DIFD	Number of line deletions on a file in last commit
DIFC	Number of chunks on a file in last commit
DIFER	For 60 records we cannot establish last commit size
SDIFI	Sum of all line insertions per file per task
SDIFD	Sum of all line deletions per file per task
SDIFC	Sum of all chunks per file per task
LFRD	Duration of the file change process measured in the number of revisions throughout the project (*)
LMP	How many days have elapsed since last modification of file in concrete svn branch
LMRP	How many days have elapsed since last modification of file across all svn branches
LMU	How many days have elapsed since last modification of file across all svn branches by user who committed this change
DAISS	Number of requirements assigned to issue (*)
DACR	Number of requirements assigned to task (*)

Source code metrics for Java and .Net classes:

WBA	Average block depth for file
WBB	Agerage complexity for file
WBC	Number of lines of code
WBD	Number of statements
WBE	Number of statements per method
WBF	Number of lines number of deepest block
WBG	Number of lines of most complex method
WBH	Maximum complexity
WBI	For 3124 records we were not able to obtain metrics from file exported by SourceMonitor

WBJ For 9 records SourceMonitor throws an exception
WBK For files that were deleted metrics were not calculated

References

[1] *The Economic Impact of Inadequate Infrastructure for Software Testing*. National Institute Of Standards & Technology, 2002.

[2] L. A. Curhan, "Software defect tracking during new product development of a computer system,"

[3] D. Huizinga and A. Kolawa, *Automated Defect Prevention: Best Practices in Software Management*. .

[4] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, New York, NY, USA, 2014, pp. 192–201 http://dx.doi.org/10.1145/2597073.2597076.

[5] "ISO 13485 Medical devices." [Online]. Available: https://www.iso.org/iso-13485-medical-devices.html.

[6] E. Alpaydin, *Introduction to Machine Learning*. The MIT Press, 2014.

[7] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*, 4 edition. Amsterdam: Morgan Kaufmann, 2016.

[8] H. Arndt, "The Java Data Mining Package - A Data Processing Library for Java," in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, 2009, vol. 1, pp. 620–621 http://dx.doi.org/10.1109/COMPSAC.2009.88.

[9] "Python Data Analysis Library — pandas: Python Data Analysis Library." [Online]. Available: http://pandas.pydata.org/. [Accessed: 30-May-2017].

[10] "SourceMonitor V3.5." [Online]. Available: http://www.campwoodsw.com/sourcemonitor.html. [Accessed: 29-May-2017].

[11] X. Yang, R. G. Kula, N. Yoshida, and H. Iida, "Mining the Modern Code Review Repositories: A Dataset of People, Process and Product," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 460–463 http://dx.doi.org/10.1109/MSR.2016.054.

[12] A. E. Hassan, "Predicting faults using the complexity of code changes," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 78–88 http://dx.doi.org/10.1109/ICSE.2009.5070510.

[13] "CKJM extended - An extended version of Tool for Calculating Chidamber and Kemerer Java Metrics (and many other metrics)." [Online]. Available: http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/. [Accessed: 29-May-2017].

[14] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empir. Softw. Eng.*, vol. 17, no. 4–5, pp. 531–577, Aug. 2012 http://dx.doi.org/10.1007/s10664-011-9173-9.

[15] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explor Newsl*, vol. 11, no. 1, pp. 10–18, Nov. 2009 http://dx.doi.org/10.1145/1656274.1656278.

[16] J. I. Khan, A. U. Gias, M. S. Siddik, M. H. Rahman, S. M. Khaled, and M. Shoyaib, "An attribute selection process for software defect prediction," in *2014 International Conference on Informatics, Electronics Vision (ICIEV)*, 2014, pp. 1–4 http://dx.doi.org/10.1109/ICIEV.2014.6850791.

[17] B. Mishra and K. K. Shukla, "Impact of attribute selection on defect proneness prediction in OO software," in *2011 2nd International Conference on Computer and Communication Technology (ICCCT-2011)*, 2011, pp. 367–372 http://dx.doi.org/10.1109/ICCCT.2011.6075151.

[18] T. M. Khoshgoftaar, K. Gao, and N. Seliya, "Attribute Selection and Imbalanced Data: Problems in Software Defect Prediction," in *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, 2010, vol. 1, pp. 137–144 http://dx.doi.org/10.1109/ICTAI.2010.27.