FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 43 (2018) No. 1
ISSN 0867-6356
DE GRUYTER OPEN
DOI: 10.1515/fcds-2018-0002
e-ISSN 2300-3405

Cost Effectiveness of Software Defect Prediction in an Industrial Project

Jaroslaw Hryszko, Lech Madeyski *

Abstract. Software defect prediction is a promising approach aiming to increase software quality and, as a result, development pace. Unfortunately, the cost effectiveness of software defect prediction in industrial settings is not eagerly shared by the pioneering companies. In particular, this is the first attempt to investigate the cost effectiveness of using the DePress open source software measurement framework (jointly developed by Wroclaw University of Science and Technology, and Capgemini software development company) for defect prediction in commercial software projects. We explore whether defect prediction can positively impact an industrial software development project by generating profits. To meet this goal, we conducted a defect prediction and simulated potential quality assurance costs based on the best possible prediction results when using a default, non-tweaked DePress configuration, as well as the proposed Quality Assurance (QA) strategy. Results of our investigation are optimistic: we estimated that quality assurance costs can be reduced by almost 30% when the proposed approach will be used, while estimated DePress usage Return on Investment (ROI) is fully 73 (7300%), and Benefits Cost Ratio (BCR) is 74. Such promising results, being the outcome of the presented research, have caused the acceptance of continued usage of the DePress-based software defect prediction for actual industrial projects run by Volvo Group.

Keywords: software defect prediction, industrial application, quality assurance, quality assurance cost

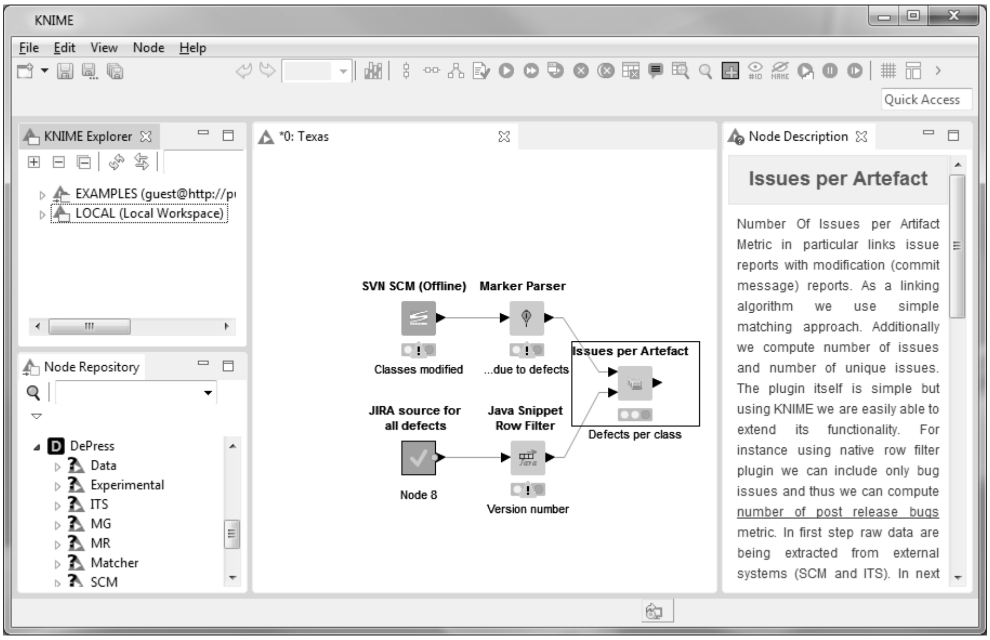
1. Introduction

The first attempts to use machine learning for software development quality assurance, were made in the early 1990's [27]. Since then, this approach has gradually improved.

*Faculty of Computer Science and Management, Wroclaw University of Science and Technology, {jaroslaw.hryszko, lech.madeyski}@pwr.edu.pl

Why then, has it not gained wider popularity in commercial projects so far? One reason is the wide variety of data necessary to perform a prediction as well as the many different sources of data which are encountered in commercial software development projects. The need to gather data from a wide range of sources, of different formats and using different access protocols or application programming interfaces (APIs) was an obstacle. Combining data with different machine learning algorithms of different characteristics was another barrier. Software defect prediction has been considered to be too complex a process, too cost and time-consuming, and there have been no known solutions for wrapping it into one, universal, defect prediction application which could be used for a different projects.

To fill this gap, Madeyski and Majchrzak [21] proposed a new, extensible (plugin-based) framework called DePress. DePress (*Defect Prediction for Software Systems*) builds upon the KNIME framework [18] and allows development of graphical workflows and uses an intuitive, user-friendly interface (see Figure 1). Being intuitive and highly customizable, DePress makes itself a perfect tool which can be conveniently utilized (thanks to its user-friendly interface and a wide range of plugins to different tools widely used in software development) in different commercial software development projects for software defect prediction. Detailed description of the DePress framework and its capabilities can be found on the DePress website [22] and in the article by Madeyski and Majchrzak [21].



Issues per Artefact

Number Of Issues per Artifact Metric in particular links issue reports with modification (commit message) reports. As a linking algorithm we use simple matching approach. Additionally we compute number of issues and number of unique issues. The plugin itself is simple but using KNIME we are easily able to extend its functionality. For instance using native row filter plugin we can include only bug issues and thus we can compute number of post release bugs metric. In first step raw data are being extracted from external systems (SCM and ITS). In next

Figure 1: DePress tool

This work is an extension of our previous conference paper [9], which investigated cost effectiveness in industrial software development project.

The extension includes, e.g., details of investigated software development project, what readers can find important for the possible reuse of the proposed approach in their research (Section 2), as well as details of a prediction model construction (new Section 3) later used to calculate potential costs when quality assurance[1] is supported by defect prediction (Section 4).

The discussion of threats to validity also has been extended (Section 5) to better sketch the limitations of the conducted empirical research, as well as discussion of results in general (Section 6), where possible result improvements and future work proposals were made.

This work extends the previous study by presenting how the prediction model was built using DePress framework, showing how values crucial for costs calculation were obtained and allowing for a wider discussion and further reproduction for example by software development practitioners in industry. This should streamline deployment of software defect prediction in general, and DePress framework in particular in other industrial settings. It is important to obtain an independent empirical evidence how useful is the proposed approach and the tool set. The more so that potential benefits of using the DePress framework [21] for defect prediction in commercial software development projects have not been investigated yet in other settings than presented in this paper. To change this, our study aimed to answer the following research questions:

RQ1: *What is the highest level of defect prediction, measured by F-measure, achievable by the DePress tool (using a default, non-tweaked configuration) in an industrial software project?*

The possible benefit varies, depending on the potential prediction effectiveness. This implies the need of first verifying what is the highest F-measure (harmonic mean of precision and recall [35]) value of the defect prediction process handled entirely by the DePress tool. DePress can be highly customizable thanks to its plugin-based architecture, as well as its open source nature. However, such adjustments can generate additional costs. Therefore, for the sake of simplicity, we decided to restrict the DePress usage only to its default set-up.

RQ2: *How cost effective is defect prediction using the DePress framework, in the default configuration, for defect prediction in an industrial software development project?*

The next step is to verify what will be the profit from the best prediction achievable using the default DePress' set-up. To achieve this, we used value of the *recall* measure (the proportion of code units predicted as defective that were actually defective [49]) corresponding to the highest F-measure value, as *recall* reflects the impact of the prediction effectiveness on the overall effort allocation strategy and possible quality assurance cost savings. This last factor as well as expected *Investment* costs then can be used to calculate *Benefit Cost Ratio* and *NetReturn* factors.

RQ3: *Will usage of the DePress framework pay off for an industrial project?*

[1]Quality assurance is a very wide notion not limited to defects handling only – it encompasses the whole software development process and relates to all project artifacts; in the context of the paper it is used in a more narrow sense which some would call quality appraisal.

To answer this question, we had to compare the costs of introducing and using the DePress based defect prediction to the potential benefits generated by its introduction. To achieve this, we used values such as return on investment (ROI) and benefit-cost ratio (BCR) [26].

The next section specifies the context of the project, while the information what kind of software processes and projects are good for the application of the concepts presented in the article, stemming from the software project under investigation, may be found at the end of Section 2.1.

2. Project Context

Volvo Group, one of the leading automotive companies, was invited to take part in this research. The primary motivation for Volvo Group's interest was to verify, if their company can use DePress and its software defect prediction to increase quality and cost-effectiveness of quality assurance (QA) in their software development projects.

Business context of the presented approach, including software project and process is summarized in Section 2.1.

2.1. Target Software Project and Process

An important criteria for selecting the proper software development project for the purpose of this research, was that such a project would be mature enough, to provide historical information which can be used as a source of training data. A special survey was conducted by Volvo Group to select the suitable candidates for initial research.

During our previous research, we recognized elements occurring in projects that hindered or prevented completion of prediction [8]. The most important of these elements are:

- Unavailability of data necessary to create a prediction model;
- Lack of possibility to link code changes with defects;
- Lack, or incorrect version, of available data;
- Inappropriate use of a version control system: avoiding atomic commits, mixing defect fixes and other changes in single commits, etc.

It is worth mentioning that the Unified Change Management (UCM), known from Rational Unified Process (RUP), unifies the activities used to plan and track project progress and the artifacts undergoing change addressing also the need for careful defect handling (e.g., separated handling of change requests resulting from defects together with their careful management).

Among few available to us for research, we selected project called Texas – which develops subsequent versions of the application under the same name. Reasons for selecting this particular project were:

- The Texas project provides all the data necessary to achieve the highest possible recall of prediction, such as:
  - Information on defects found;
  - Registered code changes as a result of defects;
  - Code metrics.
- In the Texas project, it is possible to clearly distinguish which code changes apply only to defect fixes. This is possible thanks to the practice adopted by its developers: any modification of source code is committed to the code repository with an appropriate comment.
- In the case of a modification resulting from the fix of a particular defect, a unique identifier is included in the comment. Moreover, such a feature is an example of the correct usage of a version control system.
- Information on the number of subsequent versions of the software are available, and the naming of each version is standardized.

From an organizational perspective, the Texas application is a document management tool that supports the Vehicle Type Approval (VTA) certification process for vehicle components and completed vehicles produced by Volvo Group. The VTA certificate confirms that the production samples of a design will meet specified performance standards. Key users of the Texas tool are Volvo Group's Certification Managers and engineers, as well as their brand representatives and European market companies. The certification process is crucial for the company, as availability of Volvo Group's products directly depend on it, making the reliability, development and maintenance of the Texas software highly important. Defects in the application can delay the work of certification engineers, which is unacceptable as such delays affect the scheduled dates for approval of certificate documents and publication process and, therefore have a negative impact on the date of product availability.

From a technical perspective, Texas is a Java-based application written using the Java Enterprise Edition computing platform [28]. The main development environment used for the development process is Eclipse Integrated Development Environment [44] and Apache Maven [42], its main build automation tool. To assemble a complete application, 13 different Maven projects are needed to be built by default. For the purpose of this research, 18 consecutive software versions were available, ranging from Texas version 4.0.0 to version 6.0.0, stored in a code repository managed by the Apache Subversion (SVN) version control system [43].

In the investigated project, software development process uses methodology similar to the *waterfall* model [3]. In the Texas software development process we observe the following phases[2]:

1. Requirements analysis
2. Software design

[2]The notion of "phases" is used by the IS-GDP4IT software process used in the Texas project.

3. Implementation
4. Testing
5. Deployment
6. Maintenance

Due to fact, that in considered project defects were registered and fixed only during implementation, testing and maintenance phases, only these three phases were considered in the later research.

The proposed approach can be used in projects using the *waterfall* process, as well as software processes where the aforementioned phases are iterated.

An interesting question is whether the presented approach is good for software products just having a sequence of versions each targeted to all customers (IT company's market) or if the approach is applicable also for product-lines context (customers' market). In both cases there is enough historical data to use the approach presented in the paper. Hence, we do not see obvious obstacles to use the presented approach in both cases. That said, the product-lines context, where there are many versions of a product for different customers, is more challenging than the investigated context of software products having a sequence of versions, each targeted to all customers, and thus needs further investigation to answer the question whether the presented approach may be used with good results in both contexts. Similarly, the presented approach was applied for software products distributed internally and its ability to extend to other contexts was not investigated. That said, the proposed methodology should be useful for different software products, not only these distributed internally, as long as there is possible to create an effective training dataset.

2.2. Related Work

The first publication related to an industrial application of defect prediction was published in 1997 by Khoshgoftaar et al. [13]. It was a case study of quality modeling for a very large telecommunications system. Two other publications of Khoshgoftaar and Seliya from 2004 [15] and 2005 [16] continued with the previous concept and focused on commercial data analysis, but were not applied to a real-world environment. A similar approach can be found in publications by Ostrand and Weyuker [29], Ostrand et al. [31], Tosun et al. [45], Turhan et al. [47, 48]. Examples of industrial applications of information gathered by using defect prediction can be found in publications by Wong et al. [50], Succi et al. [41] and Kläs et al. [17]. Complete cases describing the introduction of defect prediction in industrial environments were presented by Ostrand et al. [30], Li et al. [19] and Tosun et al. [46]. Unfortunately, none of the aforementioned works contain information on cost effectiveness of applied prediction techniques and tools. To the best of our knowledge, the only research focused on the cost effectiveness of software defect prediction in an industrial project, is conducted by Monden et al. [24]. However, they investigated cost effectiveness only from the acceptance testing effort perspective and do not use any quantitative measure of potential cost of quality assurance-focused work and cost of investment during the entire software life-cycle period. Thus, in our research we also followed approaches used when cost effectiveness of other than defect prediction quality assurance technique was investigated, such as Test-Driven Development return on investment research conducted by Müller and Padberg [26].

3. Construction of Defect Prediction Model

3.1. Training and Dataset Evaluation

Two factors related to the dataset used, greatly affect the quality of defect prediction:

- Size of the dataset;
- Number of defects found in the dataset's source version.

Size of the dataset is important, because in machine learning, an effectiveness of the process in most cases is proportional to the size of the dataset: the larger the dataset, the greater the efficiency of the machine learning mechanism.

Number of defects has a direct relation to the possible occurrence of a *class imbalance problem* – such as when the total number of data instances from a single class of data (in this case *defective*) is far less than the total number of another class of instances (*non-defective*). If a relatively low number of defects were detected for a relatively large application in a particular version, we should expect a large class imbalance.

Considering the above factors, we analyzed each available version for its size and number of defects registered. For the size determinant, we selected the number of separate code modules (Java classes – not to be confused with data classes). To obtain the number of modules per each version, code metrics had to be collected. Collecting code metrics has to be conducted during the process of building projects (in this case by Maven tool). To minimize impact on the Texas project team work, we decided to copy the source code and build each version, project by project, locally within the Eclipse Integrated Development Environment. To collect the metrics during the build, Eclipse Metrics 2 tool was used. Eclipse Metrics 2 is a CPL-license based Free Software tool, which works as an Eclipse IDE plugin [5] and was created as continuation of Eclipse Metrics – original metrics collection Eclipse plugin [37]. Eclipse Metrics 2 permits the collection of different kinds of code metrics (see Table 1) and exports them to an XML file. The metrics data can then be read by a dedicated DePress node (also called Eclipse Metrics) and converted into DePress' internal data format.

The best source for obtaining information about the number of defects registered in each version is a tool used for defect tracking. In the case of the Texas project, the software used for that purpose was JIRA created by the Atlassian company [2]. It means that in our research we considered all defects registered in the JIRA defect/issue tracker (and leaving any traces of code changes due to defect fixing in the SVN source code repository) regardless of their origin including requirement tests, integration tests,

Table 1: Java class-level code metrics measured by Eclipse Metrics 2 tool

Metric name	Metrics
NumberOfOverridenMethods	Total number of methods in Java class that are overridden from an ancestor class
NumberOfAttributes	Total number of Java class' attributes
NumberOfChildren	Total number of Java class' direct subclasses
NumberOfMethods	Total number of methods in Java class
DepthOfInheritanceTree	In the inheritance hierarchy, distance from Java class object
LackOfCohesionOfMethods	Cohesiveness of a Java class calculated with the Henderson-Sellers method
NumberOfStaticMethods	Total number of static methods in Java class
SpecializationIndex	Relation between number of methods and depth of inheritance tree
WeightedMethodsPerClass	Sum of the cyclomatic complexity measured for all methods in Java class
NumberOfStaticAttributes	Total number of Java class' static attributes

functional and non-functional requirements. Errors in test scripts were not taken into account – they were not registered in JIRA as Texas software defects, there were no traces of script error fixing in Texas' source code (similarly with errors in application resources). Similarly to Eclipse Metrics, JIRA also allows the export of defect data into an XML file, and that file can then be parsed by a DePress node called *Jira Offline*.

When transferred to DePress, size and defect data can be aggregated with a few simple steps using the KNIME framework capabilities. Results for each historical Texas version are presented in Table 2.

After analyzing the data presented in Table 2, we decided to use the dataset based on version 4.0.0, both for training and evaluation: the highest number of detected defects were compared to number of code modules available for analysis which made version 4.0.0 the most suitable for the above purposes. For further research, versions 5.0.0 and 6.0.0 can also be used. Other versions – with a significantly lower number of registered defects to a similar amount of modules available for analysis (defect per module ratio < 0.1) – made us assume that strong class imbalance would be observed.

Datasets for both purposes – training and validation, were constructed by split-

Table 2: Defects and number of modules per version

Version number	Defects	Modules	Defect per module ratio
4.0.0	837	744	1.125
4.0.1	19	685	0.03
4.0.2	23	673	0.03
4.1.0	45	597	0.08
4.1.1	17	546	0.03
4.2.0	54	608	0.09
4.3.0	10	660	0.02
4.3.1	8	608	0.01
4.4.0	29	616	0.05
4.5.0	11	747	0.01
5.0.0	270	744	0.36
5.1.0	15	593	0.03
5.2.0	23	606	0.04
5.4.0	35	582	0.06
5.5.0	42	582	0.07
5.6.2	10	612	0.02
5.6.3	1	612	0.002
6.0.0	462	688	0.67

ting 4.0.0 data, using a built-in DePress' function called *Stratified Sampling*, so that after the division, the ratio between code parts classified differently would be preserved in both created datasets.

3.2. Objective Variable And Class Imbalance Counteraction

For the purpose of this research, we chose two-value ("0" and "1") objective variable to distinguish the fault-prone module, where at least one defect occurred ("1"), from the fault-free modules, where no defects were observed ("0").

To counteract against any possible class imbalance, we decided to randomly remove some of the majority class instances. To follow our initial approach, we used the basic mechanisms built into DePress/KNIME by constructing a workflow as shown in

Figure 2. First, the dataset is split into two parts, by classifying rows of two different sets, depending on the objective variable value ("1" or "0"), using the *Row Splitter* KNIME node. Then, the majority class instances were reduced to reach exactly the same number of instances of minority class, achieved by random sampling done with the *Row Sampling* node. Finally, two equal size record sets were merged by *Concatenate* node to make one dataset with an equal ratio between the classes as a result.

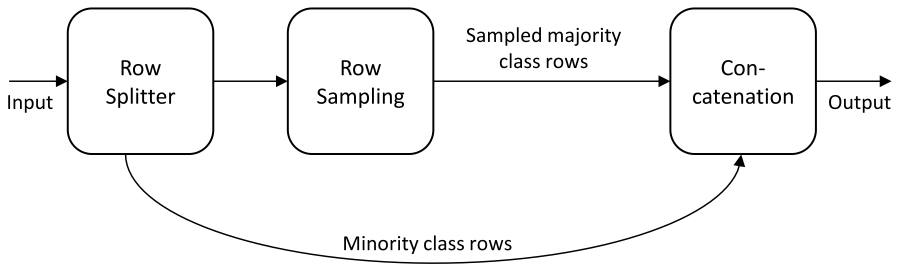


Figure 2: Class imbalance counteraction

3.3. Predictor Variables Selection

The total available dataset which can be used for machine learning, can be modified to minimize the prediction error. Various techniques can be used for this purpose, such as removing any unnecessary information, but the most commonly used technique is Feature Selection. It is used to select the most valuable code metrics from the whole set of metrics collected by the Eclipse Metrics 2 tool (Table 1). When the number of metrics is relatively small, the Feature Selection process can be based on an exhaustive search algorithm, covering all possible combinations of the available metric kinds. For each combination indicated by the algorithm, the process of prediction model creation and its validation is conducted, and then the prediction error is determined. The metrics sub-set that results in the smallest prediction error, is then selected.

When there is a large number of available metrics (features), usage of an exhaustive search algorithm is very expensive. Pendharkar [32] estimates that when an algorithm of such a type would be applied to a dataset consisting of 21 different metrics, it would take approximately 120 years of calculation by a 900 MHz RISC processor based computer to complete the Feature Selection process.

Feature Selection functionality based on the backward elimination algorithm is an integral part of the KNIME framework [18]. It consists of a data flow constructed by nodes *Backward Feature Elimination Start* and *Backward Feature Elimination End* (see Figure 3). In the loop created by these nodes, the appropriate machine learning mechanism should be placed, along with any additional supporting nodes, if needed. In Figure 3, two machine learning nodes are shown: *Random Forest Learner* and *Random Forest Predictor*. Connection between these nodes is provided to transfer the

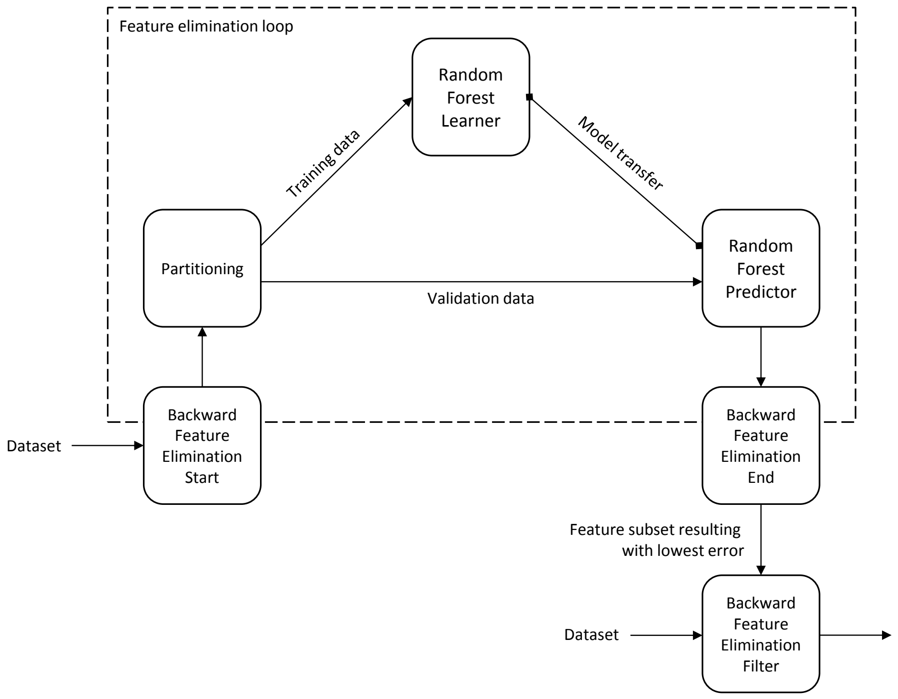

Feature elimination loop
Random Forest Learner
Training data
Model transfer
Partitioning
Random Forest Predictor
Validation data
Dataset
Backward Feature Elimination Start
Backward Feature Elimination End
Feature subset resulting with lowest error
Dataset
Backward Feature Elimination Filter

Figure 3: Depress's feature selection workflow

created model from the *Learner* to the *Predictor* node. To divide the input data for training and validation sets, the node *Partitioning* is used.

The backward elimination approach used in the DePress framework is carried out in $\left\lceil \frac{n \times (n+1)}{2} \right\rceil - 1$ iterations, where n is the total number of features (columns) in the input dataset (input table). As in this case, the total number of types of metrics collected $n = 5$, the entire process takes 14 iterations:

1. In the first iteration, the loop is executed with all features (columns): The dataset is divided into two sets; a model is created by the *Learner* node using the first set, then validated by the *Predictor* node using the second set.

2. In the next four $n - 1$ iterations, each input column is omitted once. Model creation and its validation are performed for each iteration and prediction results are collected.

3. The *Backward Feature Elimination End* node discards the column that influenced the result the least.

4. The process repeats until one feature (column) is left.

Then the *Backward Feature Elimination Filter* node is used to filter the actual dataset, using the best feature combination found as a result of the above process.

More sophisticated feature selection methods are possible, but the above-mentioned process was used in this research.

3.4. Prediction Models

Due to the fact that DePress is based on the data mining framework KNIME, various types of fault-prone module predictors can be used for the purpose of research. Since prediction results are categorical (*faulty* or *not-faulty*), we decided to test classifiers often used in software defect prediction [7, 14, 25, 38], which are available in the basic package of KNIME:

- Naive Bayes,
- Probabilistic Neural Network,
- Decision Tree.

More information about the build-in KNIME classifiers can be found in the documentation of KNIME [18].

Prediction results – modules marked as defect-prone or non-defect prone, can be compared against actual data describing defect-prone module distribution and were used to build the confusion matrix (Table 3) – a commonly used tool for performance comparison across categorical studies [7].

Table 3: Confusion Matrix

Code units	Predicted faulty	Predicted not faulty
Actually faulty	True positives (TPs)	False negatives (FNs)
Actually not faulty	False positives (FPs)	True negatives (TNs)

4. Cost Effectiveness Assessment Method

To investigate the cost effectiveness of defect prediction applied to an industrial software development project using the DePress framework, we developed the following plan to follow:

1. Development of a QA effort allocation strategy, based on defect prediction provided by DePress;
2. Analysis of actual, real-life costs of quality assurance for the selected version of the Texas project (4.0.0);
3. Building software prediction models for the chosen version;

4. Selection of the highest prediction *F-measure* and the corresponding *recall* measure;
5. Usage of an effort allocation strategy, based on the prediction effectiveness characterized by *recall*, to simulate a prediction-based quality assurance scenario;
6. Results analysis.

4.1. Quality Assurance Effort Allocation Strategy

In the case investigated (the version 4.0.0 of the Texas software), developers agreed that all the modules that caused two and more registered defects are considered to be "high risk" modules. These modules accounted for 22.4% of all modules and were responsible for 80.36% of all registered defects and the aim was to eliminate the maximum number of software defects using the available resources within a limited time period. A similar distribution of defects in the software modules was observed by different authors [6, 33, 36] and can be interpreted as the Pareto principle existence in software quality. Additionally, in 1976 Boehm argued that defect fixing costs are the more expensive the later defects are removed [4]. That observation, which is widely called Boehm's Law [6], results in another important consequence of smart quality assurance efforts allocation: the earlier the QA actions will take place, the better it is from the perspective of the software development project's budget.

Considering the above facts, we proposed a strategy which would use the prediction model to indicate as much as possible of the mentioned "high risk" software modules (22.4% in our case) responsible for most of the defects (80.36% in our case), therefore helping to integrate as much as possible the QA efforts into the implementation phase of the software development, while defect fixing cost is still relatively low. Such an approach should ideally decrease the total cost of defect fixing in the project and generate savings for the total project's budget [39].

If we denote M_{total} as the total number of testable software modules and H_{total} as the total number of discoverable defects, we can say that, in the project we analyzed, approximately $0.8H_{total}$ comes from approximately $0.22M_{total}$.

The impact of the prediction effectiveness on the overall effort allocation strategy can be reflected by using the *recall* measure (*Rec*) – the proportion of code units predicted as defective that were actually defective [49]:

$$Rec = \frac{TPs}{TPs + FNs} \tag{1}$$

We can expect that:

$$0 < Rec < 1 \tag{2}$$

Where *Rec* is the measured *recall* value corresponding to highest possible *F-measure* of defect prediction performed using the DePress framework with default settings [21]. Then, the expected number of predicted modules M_i, responsible for 80% of discoverable defects, should be:

$$M_i = 0.22 \times Rec \times M_{total} \tag{3}$$

Accordingly, we should expect that if the machine learning mechanism will be able to point out the "high risk" 22% of software modules with the measured *recall* (Rec), the number of defects which can be avoided by allocation of the best quality assurance efforts on the implementation phase, shall be:

$$H_1' = 0.8 \times Rec \times H_{total} \tag{4}$$

Number of defects expected to be detected in the implementation and maintenance phase of the project:

$$H_{2+3}' = H_{total} - H_1' \tag{5}$$

4.1.1. Return on Investment

To investigate if the DePress usage will pay off, we will use Return on Investment (ROI) [26]:

$$ROI = \frac{Benefit - Investment}{Investment} \tag{6}$$

If the investment will not pay off, ROI is negative, otherwise positive. In our evaluation of defect prediction cost-effectiveness we will focus on potential benefits that method will generate:

$$Benefit = C_{total} - C_{total}' \tag{7}$$

Where C_{total}' is the simulated total quality assurance cost in the project with defect prediction applied, and C_{total} is the actual QA cost in the project, without defect prediction.

$Investment$ is defined as the total cost of introduction of defect prediction. Moreover, $NetReturn$ is calculated as $Benefit$ reduced by $Investment$:

$$NetReturn = C_{total} - (C_{total}' + Investment) = Benefit - Investment \tag{8}$$

4.1.2. Benefit Cost Ratio

To analyze potential benefits from the usage of defect prediction, we will use the Benefit Cost Ratio (BCR) [26]:

$$BCR = \frac{Benefit}{Investment} \tag{9}$$

Values larger than 1 for the BCR mean a monetary gain from the DePress based defect prediction usage, while values smaller than 1 mean denote a loss.

4.2. Actual Project's Quality Assurance Costs

The Volvo Group policy did not allow us to publish the real costs of work invested in the project. For the purpose of research, we agreed that the man-hour cost of work by a software developer C_d will be marked as:

$$C_d = x \tag{10}$$

In that case, the average man-hour cost of work by a software tester C_t shall be, calculated according to current labor market data rates [40]:

$$C_t = 0.85x \tag{11}$$

That means, that when a tester and a developer are working together on defect fixing during the implementation and maintenance phases of the project (not in the implementation phase), the average cost per man-hour should be:

$$C_{d+t} = \frac{C_d + C_t}{2} = 0.925x \tag{12}$$

Other costs, such as infrastructure and hardware, will remain constant for the real-life and alternative (prediction-based) scenario.

Time spent on project work was traced by every team member using the JIRA tool. As a result of analysis of that data, we could obtain an average of the total time spent on fixing a single defect for each considered phase of the project (Table 4). The amount of time spent on quality assurance, together with the number of hours spent and number of defects fixed, divided by considered project phases, are shown in Table 5.

Table 4: Average defect fixing costs

Phase	1.Implementation	2.Testing	3.Maintenance
Team members involved	Developer	Developer Tester	Developer Tester
Average fixing time per one defect [hours], T	1	3	3
Assumed cost of man-hour C_{hour}	C_d	C_{d+t}	C_{d+t}
Cost per one defect $C_{defect} = T \times C_{hour}$	x	2.775x	2.775x

According to the data in Table 5, the total number of defects discovered in version 4.0.0 is:

$$H_{total} = \sum H_{phase} = 190 + 383 + 264 = 837 \tag{13}$$

Accordingly, the total quality assurance cost is:

$$C_{total} = \sum C_{phase} = 190x + 1063x + 733x = 1985x \tag{14}$$

The ratio between defects fixed in testing and those fixed during the testing and maintenance phases is:

$$\frac{H_2}{H_3} = \frac{383}{264} \approx \frac{3}{2} \tag{15}$$

Table 5: Actual resources consumed on defect fixing

Phase	1.Implementation	2.Testing	3.Maintenance
Number of defects discovered H_{phase}	190	383	264
QA cost per one defect C_{defect}	x	2.775x	2.775x
QA cost per phase $C_{phase} = H_{phase} \times C_{defect}$	190x	1063x	733x

4.3. Model Construction and Prediction

To create a prediction model that will be able to classify new source code, a machine learning process must be performed using training data already categorized respectively, and taken from an earlier version of the program, in which the area of "high risk" can be easily identified by investigating which code areas were changed most because of defect fixes. In the case of the Texas Project we used the previously described method, that all of the source code changes caused by fixing defects, were labeled with unique defect identifiers, which were recorded in the software responsible for managing defects in the project. Both, version control system, and defect tracker application used in the project make it possible to obtain the desired data by exporting it to a file. The DePress framework consists of appropriate modules allowing for conversion of the data from both those programs to a universal internal format. Log entries from the repository, take into account all changes in the source code. All recorded changes resulting from defect fixes, are recognized by the module *Marker Parser*. These entries are then compared with data from the defect management system by the module *Issues per Artefact* (Figure 4). The effect of this module's usage are statistics describing the frequency of code changes resulting from defect fixes for each of the code modules. Due to this comparison it is easy to recognize the most defective areas which are known to be "high risk". In the case investigated (version 4.0.0), as an area of "high risk", we pointed out all the modules that have to be modified to be able to fix two or more registered defects. These modules accounted for 22.4% of all modules and were responsible for 80.36% of all registered defects.

Code metrics collected for each code module were classified in accordance with the applied approach – "high risk" modules were marked as "1" and those not belonging to this group as "0". Ratio of modules labeled as "1" to those labeled as "0" was $1 : 15.52$, which indicates a class imbalance problem.

Categorized data was divided into two equal sets by stratified sampling. One of the sets was stored for validation of created prediction models, the second was used for preparation of samples based on three selected classifiers – Naive Bayes, Decision Tree and Probabilistic Neural Network.

For each classifier, four different experimental setup preparations were possible, thanks to the module-based architecture of the DePress tool:

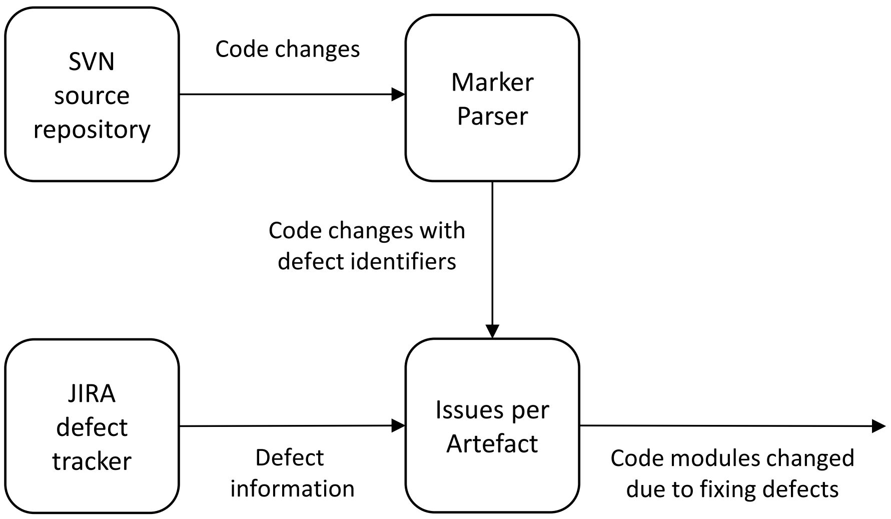


Figure 4: Workflow for combining defect and code change information

- Without feature selection, with class imbalanced dataset,
- Without feature selection, with class balanced dataset,
- With feature selection, with class imbalanced dataset,
- With feature selection, with class balanced dataset.

When needed, feature selection was carried out by KNIME's functionality presented in Section 3.3 and class balance was achieved by using mechanism presented in Section 3.2.

4.4. The Highest F-measure Value and the Corresponding Recall

Using the approach described in the previous section, defect prediction was performed and its *F-measure* collected (see Table 6) for all experimental setups, classifiers and samples. The best prediction results (the highest *F-measure* values) were obtained for the balanced class sample, slightly better with the feature selection step. Hence, we are able to answer **RQ1**: The highest *F-measure* (based on the Naive Bayes algorithm) was 0.766. The corresponding *recall* was:

$$Rec = 0.783 \tag{16}$$

Table 6: Prediction results: Recall and F-measure (in brackets) values for all experimental setups

Classifier	Without Feature Selection		With Feature Selection	
	Class Imbalance	Class Balance	Class Imbalance	Class Balance
Naive Bayes	0.409 (0.237)	0.783 (0.621)	0.318 (0.412)	0.783 (0.766)
Decision Tree	0.273 (0.279)	0.682 (0.667)	0.227 (0.357)	0.652 (0.682)
Probabilistic Neural Network	0.091 (0.167)	0.783 (0.72)	0.136 (0.24)	0.783 (0.74)

4.5. Prediction-based costs simulation

For the purpose of cost simulation in this scenario, where defect prediction is introduced to the project using the DePress framework, we assumed that:

- The total number of discoverable defects in version 4.0.0 (see Equation (13)) is a constant value;
- The defects distribution among the code is preserved;
- Average fixing cost per one defect (see Table 4) is also true for the considered scenario;
- Information on location of "high risk" software modules, with *recall Rec*, will be available in the implementation phase of the project;
- The ratio between defects fixed in testing and those fixed during the maintenance phase (see Equation (15)) is preserved.

Considering the *recall* value for best prediction achieved (characterized by the highest *F-measure* value) for version 4.0.0 as a result of the prediction models development (Equation (16)) and the total number of discovered defects in that version (Equation (13)), based on the proposed strategy (Equation (4)) we should expect, that the number of software defects which can be solved by allocation of the best quality assurance practices in the implementation phase of the project is:

$$H_1' = 0.8 \times 0.783 \times 837 = 524 \tag{17}$$

Regarding the number of defects which are expected to be found in testing and maintenance phases of the project (Equation (5)):

$$H_{2+3}' = 837 - 524 = 313 \tag{18}$$

Table 7: Simulated QA costs, with defect prediction used

Phase	1.Implementation	2.Testing	3.maintenance
Number of defects fixed H'_{phase}	524	188	125
QA cost per one defect C_{defect}	x	2.775x	2.775x
QA cost per phase $C'_{phase} = H'_{phase} \times C_{defect}$	524x	522x	347x

As we assumed that ratio in Equation (15) is preserved, the number of defects which are expected to be found in the project's testing phase is:

$$H'_2 = 313 \times 0.6 = 188 \tag{19}$$

Accordingly, the number of defects expected to be found in the maintenance phase is:

$$H'_3 = 313 \times 0.4 = 125 \tag{20}$$

Considering values H_2 and H_3, we simulated quality assurance costs assuming that the machine learning mechanism will be able to point out the "high risk" 22% of software modules with the measured *recall* (Equation (16)), and the best quality assurance efforts will be allocated to the implementation phase to avoid the calculated number of defects (Equation (17)). Results of that simulation are presented in Table 7.

Total quality assurance cost in this scenario will be:

$$C'_{total} = \sum C'_{phase} = 524x + 522x + 347x = 1393x \tag{21}$$

4.6. Cost of investment

Costs of defect prediction introduction were calculated as the sum of such elementary costs:

- Tool acquisition and installation costs,
- Training time costs,
- Data collection cost,
- Defect prediction preparation cost.

Tool acquisition and installation costs of the case investigated shall be considered as zero costs. In Volvo's organization, DePress is freely available via an internal application installation system called FAROS[3]. DePress can be ordered and installed on user's computers without any additional costs for the project.

[3]It is also available as an open source project (`http://depress.io`).

Training time costs – support of DePress and defect prediction is handled by their internal Development & Runtime Support organization, which is already pre-paid by project, whether such a tool and technology will be used or not. Due to that fact, any training cost is reduced to the cost per man-hour for a person appointed to perform defect prediction process. After measuring time spent on training of a single person, we can state that: if a person responsible for defect prediction already has minimum background in software development, she or he needs to spend a maximum of 4 hours on training, 1-2 hours of general introduction plus another 1-2 hours of training in prediction preparation.

Data collection cost is mostly the man-hour cost of exporting the proper data from data sources and code metrics generation for two selected versions. Data export will not take more that one man-hour and in most cases that activity should take no more than a few minutes. Alternatively, direct-connection nodes available in DePress can be used to get the required data in real-time. For code metrics collection purposes using *Eclipse Metrics* plugin, software need to be build locally, which include – downloading software version from the remote repository, local build and XML metric files collection. After measuring time spent on that activity, we found that it took no more than 2 man-hours.

Defect prediction preparation cost is the man-hour cost of defect prediction workflow preparation using the DePress tool. As generic workflows (for most technologies used in the organization) are pre-installed with the DePress tool, only correction actions are expected, if any. Results of time measurement say that preparation of a proper DePress workflow should not take more than one man-hour.

Summary of investment costs is presented in Table 8.

Table 8: Defect Prediction Investment Costs

Activity	Time required [hours]	Cost [man-hours]
DePress tool acquiring and installation costs	0	0
Training time costs	4	4x
Data collection cost	3	3x
Defect prediction preparation cost	1	x
TOTAL (Investment)	8	8x

4.7. Results Analysis

Here, with respect to research questions **RQ2** and **RQ3**, we summarize the results of our simulation (research question **RQ1** was answered in Section 4.4).

RQ2: *How cost effective is defect prediction using the DePress framework, in* *the default configuration, for defect prediction in an industrial software development project?*

As shown in Table 8, the expected total investment cost of the DePress tool-based defect prediction application in software development project is:

$$Investment = 8x \tag{22}$$

Benefit Cost Ratio (9) calculated using (7) and (22) values:

$$BCR = \frac{592x}{8x} = 74 \tag{23}$$

Such BCR value shows that we should expect a high monetary gain from the DePress tool usage for supporting quality assurance with defect prediction. Moreover, $NetReturn$ (Equation (8)) from the simulated defect prediction application is:

$$NetReturn = 592x - 8x = 584x \tag{24}$$

Answering research question **RQ2**, when the defect prediction application strategy proposed in Section 4.1 is applied and *recall* of the prediction model will be 0.783, such an approach can result in reduction of final QA costs by almost 30%:

$$1 - \frac{C'_{total}}{C_{total}} = 1 - \frac{1393x}{1985x} = 0.298 \tag{25}$$

Such result can be achieved only after fixing 524 defects (Equation (17)), what makes 62.6% of all the detectable defects, by effective use of quality assurance practices in the first, implementation phase of the project, on predicted "high risk" software modules. Graphical comparison of quality assurance costs – actual and simulated, with defect prediction introduced, is shown in Figure 5.

RQ3: *Will usage of the DePress framework pay off for an industrial project?*

Simulation shows, that we should expect $Benefit$ (Equation (7)) from the DePress usage in the project:

$$Benefit = 1985x - 1393x = 592x \tag{26}$$

Accordingly, expected Return on Investment (6):

$$ROI = \frac{592x - 8x}{8x} = 73 \tag{27}$$

As ROI is positive, we can state that investment will pay off.

5. Threats to Validity

In this paper, defects are not distinguished according to their severity (minor, major, etc.) and we used a fixed, average fixing time value for each defect. Omitting the severity measure in defect prediction studies is a frequent practice [23], however, it can be important when simulated QA cost calculation will be compared to real-life

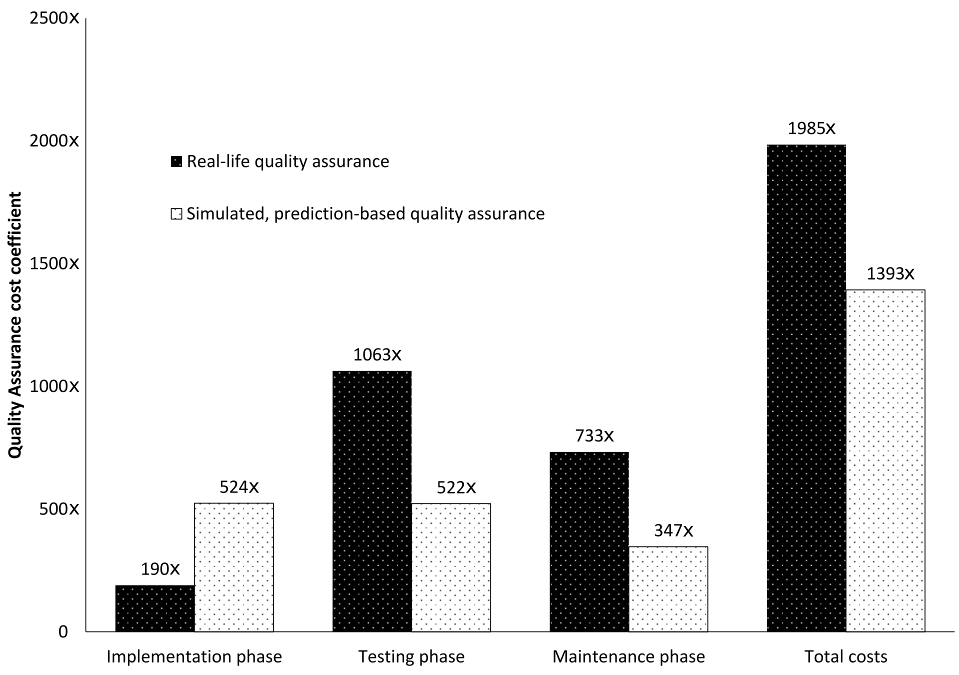


Figure 5: Costs of quality assurance in considered project: actual compared to simulated, when DePress-based defect prediction is used

values. In our simulation we assumed equal severity for each defect, which is reflected in an equal, average cost (see Table 4). However, when we apply the proposed effort allocation strategy into a real-life environment, we can deal with the situation, of when defects left undetected until the later phases of *testing* and *maintenance* that will be characterized by higher severities than defects resolved while within the implementation phase. Such a situation would negatively impact overall quality assurance costs, when DePress tool would be used for defect prediction purposes, in comparison to simulated values. Some of our assumptions, including those made in Section 4.5, may also constitute threads to validity. High CMM level (≥ 3) of the software development process within an organization would allow to fulfill them easier.

In our simulation, quality assurance cost is calculated based on the number of defects expected to be fixed by QA efforts at each considered phase, multiplied by the average cost to fix a single defect, based on actual project data. Then, simulated costs are compared to actual costs. Rahman et.al. [34] argue, that a more efficient way of comparing quality assurance efforts, when the defect prediction models are involved, is by a comparison of AUCEC (Area Under Cost Efficiency Curve) values [1]. The approach followed in our paper was motivated by the fact that the comparison of cost values (actual and simulated) is considered to be more readable by our business stakeholders.

In this paper, we do not consider any costs not traced by the JIRA system. There was no technical possibility to obtain cost information on quality assurance actions taken outside the project's team during the maintenance phase, an example being the service desk team responsible for contact with an end user, who found a new defect in the maintenance phase.

For the purpose of simulation, we used the same initial conditions as in an actual project. There were alike number of detectable defects as well as a similar ratio between defects fixed in testing phase and those fixed in the maintenance phase (5). In the actual defect prediction using the proposed effort allocation strategy, we should expect that the aforementioned conditions would be different, for example if the actual application will consider different software version and/or development project.

It is worth emphasizing that one chosen project in one company (Volvo Group) does not confirm the advantages of the presented approach from the statistical point of view.

6. Discussion and Conclusions

Cost effectiveness within the simulated scenario is strictly related to the quality of prediction when measured with recall (Rec). Based on the simulation presented, it is possible to calculate the $NetReturn$ of using a proposed quality assurance effort allocation strategy for a series of defect prediction recall values (solid line on Figure 6). In such a way, we can observe how $NetReturn$ depends on Rec in terms of the proposed QA effort allocation strategy. However, any $NetReturn$ coefficient values for Rec lower than 0.5 should not be considered, as below that point, the efficiency of defect prediction is similar to that of random guessing.

One can argue, that it is improper for one to assume, that it is possible for programmers to predict and fix 78% of all detectable defects while still in the implementation phase, only due to the fact that code parts actually responsible for these defects are known the developers, thanks to defect prediction. Please notice, that in our simulation we considered only defects actually fixed by programmers. The only difference is that in the actual project (not in simulation), more defects were found by testers and end-users during the testing and maintenance project phases, which caused higher average defect fixing costs in these phases. Nevertheless, the defects were finally fixed by the developers. Moreover, we could go a step further and modify our effort allocation strategy to use DePress for prediction model creation, aimed at finding code responsible for 100% of all detectable defects. In such a case, the relation between the expected number of defects fixed in the implementation phase, and prediction recall would be:

$$H_1' = Rec \times H_{total} \tag{28}$$

Theoretically, in such a case we can expect that the $NetReturn$ value should be even higher than calculated based on the proposed strategy (dotted line on Figure 6). However, as the proposed effort allocation strategy is based on the Pareto principle

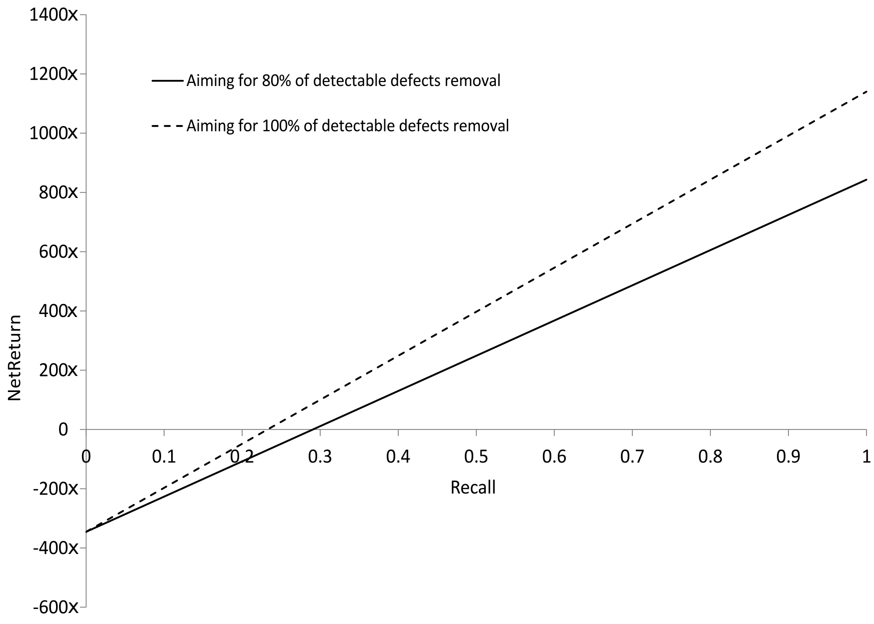


Figure 6: NetReturn coefficient depending on recall value

(only 20% of code is responsible for 80% of defects), we should consequently expect that to find and remove the remaining 20% of defects, an additional 80% of code would need to be inspected, respectively. In such a scenario, by covering even 100% of code by all necessary precautions to remove every detectable defect may cost significantly more than the actual, real quality assurance in the considered project version (4.0.0). We expect that the remaining approximately 20% of detectable defects can be effectively found and removed during further phases of the project, such as in testing phase, with better cost effectiveness, than with comprehensive quality assurance works during implementation phase, focused on finding and eliminating 100% of detectable defects.

In industrial software development projects, defect fixing consumes a significant amount of time and resources. By using the defect prediction technique, project members can obtain information on possible defect-prone elements of the software, before defects will occur, to optimally plan their quality assurance process. What is proposed in this paper, is a simple effort allocation strategy which is based on the DePress framework-driven defect prediction, the Pareto principle and the Boehm's Law, and which eliminates most of the quality assurance work during maintenance project's phase. Such an approach significantly increases quality assurance costs in implementation phase, however overall, QA costs will decrease (see Figure 5) in comparison to actual, real-life costs observed in the investigated project, as significantly less discoverable defects are left to be fixed in later phases, where, according to Boehm's Law, defect-fixing costs are considerably higher. At the same time, we need to mention the low investment costs, when a Knime-based DePress framework will be used for defect prediction purposes. Low investment costs and high recall of even simple defect prediction performed by DePress can result with high $NetReturn$ of DePress-aided quality assurance planned on a basis of the proposed effort allocation strategy. More sophisticated prediction models, especially ones using software process metrics [11, 20], may help to achieve even more impressive results. It is also worth mentioning that cross-project software defect prediction [10, 12] is sometimes used to reduce costs.

References

[1] Arisholm, E., Briand, L.C., Johannessen, E.B.: A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models. The Journal of Systems and Software 83(1), 2–17 (2010)

[2] Atlassian: JIRA Homepage (2016), `https://www.atlassian.com/software/jira/`, accessed: 2016.01.06

[3] Bell, T.E., Thayer, T.A.: Software requirements: Are they really a problem? In: Proceedings of the 2nd international conference on Software engineering. pp. 61–68. IEEE Computer Society Press (1976)

[4] Boehm, B.W.: Software Engineering. IEEE Transactions on Computers 25(12), 1226–1241 (1976)

[5] Boissier, G., Cassell, K.: Eclipse Metrics 2 Homepage (2016), `http://metrics2.sourceforge.net/`, accessed: 2016.01.06

[6] Endres, A., Rombach, D.: A Handbook of Software and Systems Engineering. Addison-Wesley (2003)

[7] Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A Systematic Literature Review on Fault Prediction Performance in Software Engineering. IEEE Transactions on Software Engineering 38(6), 1276–1304 (2012)

[8] Hryszko, J., Madeyski, L.: Bottlenecks in Software Defect Prediction Implementation in Industrial Projects. Foundations and Computing and Decision Sciences 40(1), 17–33 (2015), `http://dx.doi.org/10.1515/fcds-2015-0002`, DOI: `10.1515/fcds-2015-0002`

[9] Hryszko, J., Madeyski, L.: Assessment of the Software Defect Prediction Cost Effectiveness in an Industrial Project. In: Software Engineering: Challenges and Solutions, Advances in Intelligent Systems and Computing, vol. 504, pp. 77–90. Springer (2017), DOI: `10.1007/978-3-319-43606-7_6`

[10] Jureczko, M., Madeyski, L.: Towards Identifying Software Project Clusters with Regard to Defect Prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. pp. 9:1–9:10. PROMISE '10, ACM,

New York, USA (2010), http://dx.doi.org/10.1145/1868328.1868342, DOI: 10.1145/1868328.1868342

[11] Jureczko, M., Madeyski, L.: A Review of Process Metrics in Defect Prediction Studies. Metody Informatyki Stosowanej 30(5), 133–145 (2011), http://madeyski.e-informatyka.pl/download/Madeyski11.pdf

[12] Jureczko, M., Madeyski, L.: Cross–project defect prediction with respect to code ownership model: An empirical study. e-Informatica Software Engineering Journal 9(1), 21–35 (2015), http://dx,doi.org/10.5277/e-Inf150102, DOI: 10.5277/e-Inf150102

[13] Khoshgoftaar, T.M., Allen, E.B., Hudepohl, J.P., Aud, S.J.: Application of Neural Networks To Software Quality Modelling Of a Very Large Telecommunications System. IEEE Transactions on Neural Networks 8(4), 902–909 (1997)

[14] Khoshgoftaar, T.M., Pandya, A.S., Lanning, D.L.: Application of Neural Networks for Predicting Faults. Annals of Software Engineering 1(1), 141–154 (1995)

[15] Khoshgoftaar, T.M., Seliya, N.: Comparative Assessment of Software Quality Classification Techniques: An Empirical Case Study. Empirical Software Engineering 9(3), 229–257 (2004)

[16] Khoshgoftaar, T.M., Seliya, N.: Assessment of a New Three-Group Software Quality Classification Technique: An Empirical Case Study. Empirical Software Engineering 10(2), 183–218 (2005)

[17] Kläs, M., Nakao, H., Elberzhager, F., Münch, J.: Predicting Defect Content and Quality Assurance Effectiveness by Combining Expert Judgment and Defect Data-A Case Study. In: Proceedings of the 19th International Symposium on Software Reliability Engineering. pp. 17–26 (2008)

[18] KNIME.COM AG: KNIME Framework Documentation (2016), https://tech.knime.org/documentation/, accessed: 2016.11.06

[19] Li, P.L., Herbsleb, J., Shaw, M., Robinson, B.: Experiences and Results from Initiating Field Defect Prediction and Product Test Prioritization Efforts at ABB Inc. In: Proceedings of the 28th International Conference on Software Engineering. pp. 413–422 (2006)

[20] Madeyski, L., Jureczko, M.: Which Process Metrics Can Significantly Improve Defect Prediction Models? An Empirical Study. Software Quality Journal 23(3), 393–422 (2015), http://dx.doi.org/10.1007/s11219-014-9241-7, DOI: 10.1007/s11219-014-9241-7

[21] Madeyski, L., Majchrzak, M.: Software Measurement and Defect Prediction with DePress Extensible Framework. Foundations and Computing and Decision Sciences 39(4), 249–270 (2014), http://dx.doi.org/10.2478/fcds-2014-0014, DOI: 10.2478/fcds-2014-0014

[22] Madeyski, L., Majchrzak, M.: ImpressiveCode DePress (Defect Prediction for software systems) Extensible Framework (2016), `https://github.com/ImpressiveCode/ic-depress`

[23] Menzies, T., Jalali, O., Hihn, J., Baker, D., Lum, K.: Stable Rankings for Different Effort Models. Automated Software Engineering 17(4), 409–437 (2010)

[24] Monden, A., Shinoda, S., Shirai, K., Yoshida, J., Barker, M., Matsumoto, K.: Assessing the Cost Effectiveness of Fault Prediction in Acceptance Testing. IEEE Transactions on Software Engineering 39(10), 1345–1357 (2013)

[25] Moser, R., Pedrycz, W., Succi, G.: A Comparative Analysis of The Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In: Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on. pp. 181–190 (2008)

[26] Müller, M.M., Padberg, F.: About the Return on Investment of Test-Driven Development. In: International Workshop on Economics-Driven Software Engineering Research EDSER-5. pp. 26–31 (2003)

[27] Munson, J.C., Khoshgoftaar, T.M.: The Detection of Fault-Prone Programs. IEEE Transactions on Software Engineering 18(5), 423–433 (1992)

[28] Oracle Corporation: Java EE Homepage (2016), `http://www.oracle.com/technetwork/java/javaee/overview/index.html`, accessed: 2016.01.06

[29] Ostrand, T.J., Weyuker, E.J.: The Distribution of Faults in a Large Industrial Software System. SIGSOFT Software Engineering Notes 27, 55–64 (2002)

[30] Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Predicting the Location and Number of Faults in Large Software Systems. IEEE Transactions on Software Engineering 31(4), 340–355 (2005)

[31] Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Programmer-Based Fault Prediction. In: Proceedings of the Sixth International Conference on Predictive Models in Software Engineering. pp. 1–10 (2010)

[32] Pendharkar, P.C.: Exhaustive and Heuristic Search Approaches For Learning a Software Defect Prediction Model. Engineering Applications of Artificial Intelligence 23, 34–40 (2010)

[33] Pressman, R.: Software Engineering: A Practitioner's Approach. McGraw-Hill (2010)

[34] Rahman, F., Sammer, K., Barr, E.T., Devanbu, P.: Comparing Static Bug Finders and Statistical Prediction. In: Software Engineering, 2014. ICSE '14. ACM/IEEE International Conference on. ACM (2014)

[35] Rijsbergen, C.J.V.: Information Retrieval. Butterworth-Heinemann Newton (1979)

[36] Rizwan, M., Iqbal, M.: Application of 80/20 Rule in Software Engineering Waterfall Model. In: Proceedings of the International Conference on Information and Communication Technologies '09 (2009)

[37] Sauer, F.: Eclipse Metrics Homepage (2016), `http://metrics.sourceforge.net/`, accessed: 2016.01.06

[38] Selby, R.W., Porter, A.: Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis. IEEE Transactions on Software Engineering 14(12), 1743–1756 (1988)

[39] Slaughter, S.A., Harter, D.E., Krishnan, M.S.: Evaluating The Cost of Software Quality. Communications of the ACM 41(8), 67–73 (1998)

[40] Source of Information on Salaries in Poland (2015), `http://wynagrodzenia.pl/`, accessed: 2015.02.28

[41] Succi, G., Pedrycz, W., Stefanovic, M., Miller, J.: Practical Assessment of the Models for Identification of Defect-Prone Classes in Object-Oriented Commercial Systems Using Design Metrics. Journal of Systems and Software 65(1), 1–12 (2003)

[42] The Apache Foundation: Apache Maven Homepage (2016), `https://maven.apache.org/`, accessed: 2016.01.06

[43] The Apache Foundation: Apache Subversion Homepage (2016), `https://subversion.apache.org/`, accessed: 2016.01.06

[44] The Eclipse Foundation: Eclipse IDE Homepage (2016), `https://eclipse.org/`, accessed: 2016.01.06

[45] Tosun, A., Bener, A., Turhan, B., Menzies, T.: Practical Considerations in Deploying Statistical Methods for Defect Prediction: A Case Study within the Turkish Telecommunications Industry. Information and Software Technology 52(11), 1242–1257 (2010)

[46] Tosun, A., Turhan, B., Bener, A.: Practical Considerations in Deploying AI for Defect Prediction: A Case Study within the Turkish Telecommunication Industry. In: Proceedings of the Fifth International Conference on Predictor Models in Software Engineering. p. 11 (2009)

[47] Turhan, B., Kocak, G., Bener, A.: Data Mining Source Code for Locating Software Bugs: A Case Study in Telecommunication Industry. Expert Systems with Applications 36(6), 9986–9990 (2009)

[48] Turhan, B., Menzies, T., Bener, A., Stefano, J.D.: On the Relative Value of Cross-Company and within-Company Data for Defect Prediction. Empirical Software Engineering 14(5), 540–578 (2009)

[49] Witten, I.H., Frank, E., Hall, M.A.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann (2005)

[50] Wong, W.E., Horgan, J., Syring, M., Zage, W., Zage, D.: Applying Design Metrics to Predict Fault-Proneness: A Case Study on a Large-Scale Software System. Software: Practice and Experience 30(14), 1587–1608 (2000)

This paper is a revised and extended version of work originally presented at the 18th KKIO Software Engineering Conference, 15-17 September 2016, Wroclaw, Poland
Received 12.06.2017, Accepted 09.01.2018