1 INTRODUCTION

One of the first defect prediction models was suggested almost 40 years ago [12]. Nonetheless, the industrial usage of such methods is still very limited. The researchers advocate the benefits regarding savings in testing process without compromising on software quality, but the arguments are ignored by project managers and hence industrial application remains marginal. Let us look closer at the phenomena as it does not seem to be reasonable at first glance.

We will start with introducing the reader to the concept of a defect prediction model. In order to avoid unnecessary complications and to keep things as simple as possible, the defect prediction model will be considered as a black box. First of all the software project must be divided into smaller parts, let us call them software artefacts. Each of those artefacts is described by a set of features which we call software metrics. The values of the software metrics that describe a given software artefact create a vector of numbers (there are exceptions to the rule as there are software metrics measured on the nominal scale). The vector is then used as defect prediction model input. The output is a prediction with regard to the faultiness of the software artefact.

The most important benefit of defect prediction concerns savings in testing process. The model points out which software artefacts are defect prone and hence makes it possible to allocate all testing resources to the other ones. According to Hall et al. [11] the majority of defect prediction model performs binary classification. Let us suppose that the model predicted 40% of artefacts to be defect prone and according to project's constraints only 30% of them could be covered by the testers before deadline. Which ones should be chosen then? Furthermore, the defect prediction models treat all software artefacts equally and this is not the case in test management [26]. In each software system there are critical functionalities, which just cannot fail, and features of second importance that of course should work correctly, however when do not, the loss is not great. This is knowledge that shall not be ignored when deciding what to test.

Other significant obstacle in the adoption of the defect prediction models concerns the object of prediction, i.e. the software artefact. Till now we have not defined what exactly this term represents. As a matter of fact there is no uniform definition. Different researchers investigated different objects of prediction. According to the Hall et al. [11] the majority of studies are focused on making predictions for classes and files, but there are promising results for bigger objects as well. Nonetheless, the important thing is that the type and size of object of prediction can create challenges when implementing a defect prediction model in a real software development. Let us start with the small ones. The prediction may be focused on a method, class or file. In such cases the area of application concerns unit tests. The testers are given with information about which parts of the developed system are defect prone and hence should be tested and which parts not necessarily. Unfortunately, such approach does not correspond with good
practices of testing as the unit test are recom-
mended to make as high code coverage as possi-
ble, e.g. [2]. Furthermore, the studies on defect
prediction does not consider the return of invest-
ment (according to the author's knowledge there
is only one preliminary study on this topic [3]
and some theoretical considerations [16]) and
there could be a really big difference between
the costs of preparing unit tests depending of the
complexity and size of the object of test. More-
ever, the return of investment issue is also rele-
vant for the bigger objects of prediction. A mod-
ule or a compilation unit, e.g. a component or a
binary file are examples of such object. And in
the case of such objects the practical implement-
ation is not trivial too. A compilation unit usu-
ally covers a bunch of functionalities and due to
its size it could be extremely risky to drop the
tests on a one. The test manager typically makes
decisions with regard to what to test and what
not on the level of test cases. It is rather chal-
lenging to make it for certain method and class-
es. It could be done for units of compilation but
then we have a number of test cases for one unit
and removing all of them from the test plan
could be far away from optimal. For well-de-
defined test cases it should be straightforward to
map them on requirements that are usually ex-
pressed using use cases or user stories. Unfortu-
nately, such objects of prediction are rarely in-
vestigated. Similar to this topic are studies fo-
cused on predicting re-opens i.e. predicting
whether a defect has been correctly fixed (use
cases and user stories are other types of issues
than the defects but have a lot of similarities in-
cluding the workflow and hence the prediction
of re-opens could cover them as well). If not, the
defect should be carefully tested and possibly re-
turned to the developer for further correction.
For correctly resolved defects the aforemen-
tioned efforts could be spared.

The rest of this paper is organized as follows.
The next Section briefly presents the most im-
portant theoretical findings of PhD thesis [14]
that is focused on defect prediction (during its
creation the Statistica tool was used). Section 3
discusses an industrial application in the field of
predicting re-opens of defects. Other works that
investigate the prediction of re-opens are given
in the Section 4. Section 5 closes the paper with
conclusions.

2 THE EXPERIMENTS AND THE
FINDINGS

This paper summarizes PhD thesis [14]. Mainly
it is focused on the practical application, but
takes into account the theoretical experiments as
well. There are two areas of investigation, both
of them concerns empirical studies conducted on
data collected from industrial, open-source and
academic software projects. In all experiments
the tests are conducted on significance level \( \alpha
=.05 \). All the analysis were done using STATIS-
TICA 8.0. The investigated data sets (available
online at: http://purl.org/MarianJureczko/Metric-
sRepo) consists of information about defects and
following software metrics:

- CK metrics suite [6],
- QMOOD metrics suite [1],
- quality oriented extension of the CK metric
  suite [29],
- Martin's coupling metrics [23],
- LCOM3 [13], the number of lines of code
  (LOC) and McCabe’s Cyclomatic Complexity
  [24],
- Process metrics (described in next
  subsection).

2.1 Identifying factors that improve prediction
quality

The first type of investigations analysed which
factors (mainly process metrics) improve the
prediction quality when added to the set of met-
rics which frequently are used as independent
variables. The set of metrics consists listed
above metrics.

The relationships between the factors and the
number of defects were empirically explored. A
simple defect prediction models were built on
the basis of the frequently used metrics. With
those models it was possible to build advanced
defect prediction models by introducing particu-
lar factor. Two different methods of introducing
the investigated factor were considered. As a re-
sult, it was possible to compare the simple and
the advanced models and answer the question
whether the introduction of the factor improved
the adequacy of the predictions. Statistical meth-
ods were used to evaluate the significance of
that improvement. For the sake of simplicity no
sophisticated methods were used to build the models, but the ordinary stepwise linear regression. It is also noteworthy that the construction of the models made use solely of the data which were historically older than the ones used in prediction (model evaluation). For example, the model built on the data from the release \( i \) has been used to make predictions in version \( i+1 \). The data from \( i \)-th release is usually (or at least may be) available during the development of \((i+1)\)-th release. Partial results of those experiments were originally reported in [21] and [15].

Following factors were investigated:

- **Number of Revisions (NR)**. The NR metric represents the number of revisions of a given Java class that have been committed during the development of the investigated version of the software system.

- **Number of Distinct Committers (NDC)**. The NDC metrics count the number of distinct authors, usually developers, who had been committing their changes in a given Java class during the development of the investigated version of the software system.

- **Number of Modified Lines (NML)**. The value of the NML metrics is equal to the sum of all lines of source code that have been added or removed in a given Java class in all revisions, which have been committed during the development of the investigated version of the software system.

- **Is New (IN)**. It is a binary metric that shows whether the given class existed in the previous version of the investigated system or is a new one.

- **Number of Defects in Previous Version (NDPV)**. The NDPV metric counts the number of defects that have been fixed in a given class in at least one of the revisions, which have been committed during the development of the investigated version of the software system.

- **Number of Evening Revisions (NER)**. The NER metric represents the number of revisions of a given java class that were committed during the development of the investigated version of the software system, but only the revisions are counted that were committed “just before end of the work”. The company culture were analysed in order to find in which hours the developers usually work and an hour that is close to the end of the working-day were identified. The NER metric counts only those revisions, which were committed after the hour.

- **Number of Pre-code-freeze Revisions (NPR)**. The NPR metric represents the number of revisions of a given java class that were during the development of the investigated version up to three weeks before the “code-freeze” phase. The metric was calculated only for the projects that use the ‘code-freeze’ phase.

Table 1. Results of identification of factors that improve prediction quality. ‘↗’ - improvement in prediction; ‘↘’ - deterioration in prediction; ‘?’ - too small difference to obtain statistically significant results; '?' - too few samples to obtain statistically significant results.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Method I</th>
<th>Method II</th>
</tr>
</thead>
<tbody>
<tr>
<td>NR</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NDC</td>
<td>↗</td>
<td>-</td>
</tr>
<tr>
<td>NML</td>
<td>↗</td>
<td>-</td>
</tr>
<tr>
<td>IN</td>
<td>not applicable</td>
<td>↗</td>
</tr>
<tr>
<td>NDPV</td>
<td>-</td>
<td>?</td>
</tr>
<tr>
<td>NER</td>
<td>↗?</td>
<td>?</td>
</tr>
<tr>
<td>NPR</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>WMC</td>
<td>not applicable</td>
<td>-</td>
</tr>
<tr>
<td>LOC</td>
<td>not applicable</td>
<td>↗</td>
</tr>
</tbody>
</table>

The results are presented in Tab. 1. Three of the investigated factors significantly improved the prediction quality. Those are NDC, NML and LOC. One of the investigated metrics, i.e. IN, significantly worsened the prediction. The result depends on the method of introducing the investigated factor. In the case of NDC and NML it was enough to add the metric to the set of independent variables whereas LOC should be used to split the samples into two subsets: samples with small LOC values and samples with large LOC values (details about this procedure are given in [14]).

### 2.2 Identifying clusters of projects with regard to cross-project defect prediction

The goal of this type of experiments was to find such clusters that there is a certain defect prediction model which can be successfully used for all projects that belong to the cluster. Those experiments are motivated by the possibility of making savings in model training. With well-defined clusters it is enough to identify to which
one a project belong and use for prediction the already trained model that is assigned to the cluster.

Two different experiments were conducted. In the first one the clusters were defined apriori according to code ownership model, i.e. industrial, open-source and academic. The second one employed data mining techniques for cluster identification. In both experiments a model trained on the investigated cluster was compared with reference models. When the cluster model provides significantly better predictions for the cluster projects, we can conclude that the cluster exists. Preliminary results of those experiments were reported in [17] and [18]. The experiments were conducted using the same data sets as the ones mentioned in the previous subsection.

Table 2. Differences between clusters defined according to code ownership model.

<table>
<thead>
<tr>
<th></th>
<th>Non-industr.</th>
<th>Open-source</th>
<th>Academic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Industrial</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open-source</td>
<td>↗</td>
<td>(p=.06)</td>
<td></td>
</tr>
<tr>
<td>Academic</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The results of the first type of experiments are presented in Tab. 2. Relevant improvements were found in the case of the open-source projects. The model trained on the cluster gave significantly better predictions for open-source projects than the reference models (however, in the case of the ‘not-open’ reference model the difference is significant on α=.1 as p=.06) and hence it could be stated that such cluster exists from the defect prediction point of view.

The clusters identified using data mining techniques are evaluated in Tab. 3. The clusters with names following pattern ‘n-th of N’ were obtained using the k-means method. The assumed number of clusters was equal N. The later ones are a result of applying Kohonen Neural Network.

The obtained results show that defining a useful cluster is possible but could be very challenging. Most of the investigated clusters gave very moderate predictions. Only in three cases a statistically significant improvement was found and hence only those three clusters can be acknowledged as existing ones. Recently, this issue has been addressed by Menzies et al. [25] with a novel and promising approach.

Table 3. Evaluation of clusters identified using data mining techniques.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st of 2</td>
<td></td>
</tr>
<tr>
<td>2nd of 2</td>
<td></td>
</tr>
<tr>
<td>1st of 4</td>
<td></td>
</tr>
<tr>
<td>2nd of 4</td>
<td></td>
</tr>
<tr>
<td>3rd of 4</td>
<td></td>
</tr>
<tr>
<td>4th of 4</td>
<td></td>
</tr>
<tr>
<td>Industrial A</td>
<td></td>
</tr>
<tr>
<td>Industrial B</td>
<td></td>
</tr>
<tr>
<td>Mixed</td>
<td></td>
</tr>
<tr>
<td>Open</td>
<td></td>
</tr>
</tbody>
</table>

3 AN INDUSTRIAL APPLICATION – PREDICTING RE-OPENS

The introduction section pointed out a number of challenges that often prevent the installation of defect prediction model in an industrial environment. Both of the described in previous section experiment types suffer from the aforementioned challenges and are hardly installable without structural changes. When confronted with business needs, it could be indicated that the much more desired and applicable thing is re-opens prediction. This section describes an industrial application regarding predicting re-opens that was originally presented in the mentioned earlier PhD thesis [14].

The predictions were used for optimizing human resources in a testing process that suffered from a large number of bug fix verification requests. The process is depicted on Fig. 1. The number of requests and the efforts they created was too big to be addressed with the available resources. Simultaneously a great fraction of bug fixes were resolved correctly and hence the tests did not result in re-opening them. Since there were not enough resources to handle all requests it become critical to select the right issues to test. And this is where the re-opens prediction comes handy.
Typical issue handling process looks like the one presented on Fig. 1. When an issue (usually a defect, commonly called a bug) is identified, it should be described and reported. In the second step the issue is assigned to a software developer who is fixing it. A fixed issue should be tested in order to verify whether it has been done correctly. If not, the issue is re-opened and again assigned to a software developer. Otherwise the fix could be released to the customer. Most of the issues are fixed correctly in the first run (Shihab et al. [28] reported that it is 6.9% to 35.7% in different projects), and hence it is theoretically possibly to save a lot of efforts by not testing the correctly solved ones but releasing them instantly. Unfortunately, without tests it is challenging to guess whether a fix is correct. However, a prediction model can give plausible tips. Thus, with a prediction model the ‘3. Test the fix’ step can be skipped for a significant fraction of issues.

The investigated project is a custom-built enterprise solution that has been already developed by international team of 30+ developers and installed in the customer environments. It belongs to the insurance domain but implements different feature sets on top of Java-based frameworks.

3.1 Independent variables

The used in experiment independent variables could be categorised into four dimensions according to the type of artefact or software development process they measure:

- **static code metrics of the source code changed to fix an issue** - the metrics listed in Sec. 2 were used and calculated for the classes that were modified by a software developer during the fix preparation;

- **issue report**:  
  - CRNumber – identifier of the feature (use case) that was closely related to the issue,
  - Keywords – list of keywords assigned to the issue,
  - Priority – the priority assigned to the issue
  - Estimated effort – the effort that was estimated for solving the issue,
  - Real effort – the effort that in fact was committed to solve the issue,
  - Product – subsystem that contains the issue
  - Component – a module that contains the issue, module is a part of the subsystem,
  - NoOfAttachments – number of attachments assigned to the issue,

- **developer activities**:  
  - NoOfAssignedTo – number of different software developers that were assigned to this issue before it was fixed,
  - NoOfStatuses – number of times the issue status was changed,
  - NoOfCommentsBeforeTests – number of comments assigned to the issue before it was resolved,
  - NoOfTargetMilestones – number of times the issue release date was postponed,
  - TimeToFirstAssignedTo – time between reporting the issue and assigning it to a software developer,

- **bug fix**:  
  - AllRevisions(Java) – the sum of all revisions of all Java files that were modified during solving the issue,
  - MaxRevisions(Java) – number of revisions of this Java file which was modified the greatest number of times,
MeanAuthor(Java) – number of different authors (software developers) per modified Java file,
MaxAuthor(Java) – number of different authors (software developers) of the Java file that was modified the greatest number of times
AllRevisions(XML) – the sum of all revisions of all XML files that were modified during solving the issue,
MaxRevisions(XML) – number of revisions of this XML file which was modified the greatest number of times,
MeanAuthor(XML) – number of different authors (software developers) per modified XML file,
MaxAuthor(XML) – number of different authors (software developers) of the XML file that was modified the greatest number of times,
NoOfPropertiesFiles – number of properties files that were modified during solving the issue,
NoOfSqlFiles – number of SQL files that were modified during solving the issue,
NoOfXmlFiles – number of XML files that were modified during solving the issue,
NoOfJavaFiles – number of Java files that were modified during solving the issue.

3.2 Results
A great number of different metrics were used. Furthermore, some of them are measured on the nominal scale. Additionally there was a requirement for high readability of the reasoning mechanism (i.e. it was important to show how a conclusion was obtained). In consequence the prediction model was built using decision trees, namely C&RT and CHAID. The reasoning process in a decision tree is straightforward, the nominal variables are supported and it is easy to remove the least relevant independent variables and hence limit the number of them.

The collected data was divided into training set which consisted of 80% of available samples and testing set – the remaining 20%. The training test was used to build the model and the testing set was used to evaluate the performance of the model. The model constructed using the CHAID method gave correct predictions for 69% of testing samples, whereas the model that used C&RT was correct in 73.5% of testing samples. Thus the later one was installed and used in the project.

The obtained results are encouraging, but not perfect. The model cannot be used as an oracle since it is wrong in every fourth issue on average. Nonetheless, it is still much better than blind guessing, especially in the case of limited resources in the testing process. When it is not possible to test every issue, presumably the best solution is to use such prediction model.

4 RELATED WORK
Defect prediction is an object of very intensive research. It is hardly possible to mention all relevant works in this area. Fortunately, there are some reviews that briefly go through the most important papers, e.g. [5], [19] and [22]. This section does not take into consideration all defect prediction works, but is strictly focused on predicting re-opens.

Zimmermann et al. [30] characterized when bug reports are re-opened by using the Microsoft Windows operating system project as an empirical case study. It was a mixed-methods approach. The primary reasons for re-opens was categorised based on a survey of 358 Microsoft employees. Then the results were reinforced with a large-scale quantitative study of Windows bug reports. Factors related to bug report edits and relationships between people involved were considered and a statistical models was built to describe the impact of various metrics on re-opening bugs ranging from the reputation of the opener to how the bug was found.

Caglayan et al. [4] extracted issue activity data from a large release of an enterprise software product. Four dimensions were considered, i.e. developer activity, issue proximity network, static code metrics of the source code changed to fix an issue, issue reports and fixes as possible factors that may cause issue re-opening. Logistic regression models were built on the data in order to identify key factors leading issue re-opening. The results were complemented with a survey regarding these factors with the QA Team of the
product. The obtained results indicated that centrality in the issue proximity network and developer activities are important factors in issue re-opening. Furthermore, the quantitative findings of the study suggested that issue complexity and developers workload play an important role in triggering issue re-opening.

Shihab et al. [28] studied and constructed a prediction model for re-opened bugs through a case study on three large open source projects namely Eclipse, Apache and Open Office. Four dimensions were taken into consideration: the work habits dimension (e.g., the weekday on which the bug was initially closed), the bug report dimension (e.g., the component in which the bug was found), the bug fix dimension (e.g., the amount of time it took to perform the initial fix) and the team dimension (e.g., the experience of the bug fixer). Decision trees were built using the aforementioned factors that aim to predict re-opened bugs. Top node analysis was employed to determine which factors are the most important indicators of whether or not a bug will be re-opened. A combination of the aforementioned dimensions resulted in a model that can achieve a precision between 52.1–78.6 % and a recall in the range of 70.5–94.1 % when predicting whether a bug will be re-opened.

5 DISCUSSION AND CONCLUSIONS

The paper summarizes most important theoretical findings originally reported in [14]. Experiments regarding defect prediction that were conducted on a wide range of projects are described. The results show that there are factors that could be added to a static code metrics based defect prediction model and significantly improve the prediction quality, i.e. NDC (Number of Distinct Committers), NML (Number of Modified Lines) and LOC (Lines Of Code). The improvements are statistically significant on the significance level .05. Furthermore, it has been proven that there exist such clusters that a defect prediction model trained for the cluster performs well for all cluster members. A number of clusters were identified based on experience and using data mining techniques and for some of them the aforementioned feature has been statistically proven. The finding traces a solution for cross-project defect prediction. All the experiments gave promising results that encourages further research with regard to optimize the prediction performance and enable cross-project defect prediction. The experiments were conducted using data from real software projects. All the collected data is published online (http://purl.org/MarianJureczko/MetricsRepo) and available for other researchers and practitioners. The data acquisition process required development of a number of tools. All of them were later gathered in one system called QualitySpy [20].

The third section of this paper is devoted to the practical application of defect prediction models. An industrial case study regarding predicting re-opens of issues is described. Unfortunately, the description does not fully correspond with the guidelines for conducting and reporting case study in software engineering [27] as it was conducted under time pressure (there was tight schedule in the investigated project and the prediction model was needed right away) and limitations regarding publication of experiment details. Although, according to author knowledge, it was the first documented installation of a model for predicting re-opens. All the reported in Section 4 works were prepared later but presumably independently as the industrial application description was published only in polish. The presented in this paper industrial case study on re-opens is not as comprehensive as the related works. Only one project was investigated and only decision trees were considered as the prediction technique. It is a consequence of industrial reality and its constraints. It also was not possible to use others experiences and findings as the related works had not been published yet. Therefore, there are a number of possibilities for further research. First of all the number of investigated projects and employed prediction techniques could be increased. Besides, the experiment procedure could be more comprehensive, e.g. using the 10-fold cross validation would improve the matters. It might be also very interesting to confront the results with the findings of other researchers. The studies on defect prediction are usually theoretical, hence it is relevant to keep the further research as close to the indus-
try as possible, especially as the re-opens prediction is relatively easy to adopt in real world projects.

REFERENCES