Report on Evaluation Experiments Using Different Machine Learning Techniques for Defect Prediction

Marios Grigoriou
Dept. of Computer Science
Western University
London, ON. Canada
mgrigori@uwo.ca

Kostas Kontogiannis
Dept. of Computer Science
Western University
London, ON. Canada
kostas@csd.uwo.ca

Alberto Giammaria
IBM
IBM Austin Laboratory
Austin TX. USA
agiammaria@us.ibm.com

Chris Brealey
IBM
IBM Toronto Laboratory
Toronto ON. Canada
cbrealey@ca.ibm.com

***Abstract*—With the emergence of AI, it is of no surprise that the application of Machine Learning techniques has attracted the attention of numerous software maintenance groups around the world. For defect proneness classification in particular, the use of Machine Learning classifiers has been touted as a promising approach. As a consequence, a large volume of research works has been published in the related research literature, utilizing either proprietary data sets or the PROMISE data repository which, for the purposes of this study, focuses only on the use of source code metrics as defect prediction training features. It has been argued though by several researchers, that process metrics may provide a better option as training features than source code metrics. For this paper, we have conducted a detailed extraction of GitHub process metrics from 150 open source systems, and we report on the findings of experiments conducted by using different Machine Learning classification algorithms for defect proneness classification. The main purpose of the paper is not to propose yet another Machine Learning technique for defect proneness classification, but to present to the community a very large data set using process metrics as opposed to source code metrics, and draw some initial interesting conclusions from this statistically significant data set.**

***Index Terms*—Machine learning, software maintenance, defect-proneness, experiments, data set, open source systems.**

I. Introduction

The problem of classifying a file as failure prone or not has attracted the attention of the software engineering community early on. Early approaches focused on the use of software metrics to compute maintainability and software health indexes [19] [20]. These approaches were based on the compilation of linear or non-linear formulas to yield maintainability indexes which were assumed to be associated with the overall health of a component or a system. In this respect, the assumption was that a higher maintainability index would indicate a software component (function, method, file, or module) that has a low probability of exhibiting a failure. As research progressed in this field, the software engineering community experimented with approaches focusing on the static and dynamic analysis as well as the analysis of project data, such as the number, type and time interval between bug fixes [24] [25]. These approaches utilized statistical analyses and heuristics to experimentally yield predictions related to the fault-proneness of a software component. However, over the past few years, research in this area has decisively shifted towards the use of Machine Learning *(ML)* techniques. These techniques aim to first identify a collection of source code related and process related features which can serve as classiffiers for fault-proneness, and second apply these features for training *ML* models using a variety of *ML* algorithms (see Section II). Once such models are trained they can be used to classify whether a newly seen software component is defect prone or not.

The challenge that arises using such *ML* techniques is that they yield models which perform as black boxes and do not provide any explanation on how their results have been reached, as they are purely dependent on the training data set provided, and the *ML* algorithm used. Another challenge that arises is when *ML* models are trained on source code metrics alone. Large software systems are rarely implemented using a single programming language and are often composed of a collection of different frameworks, configuration scripts and dynamically linked components. That makes the extraction of accurate source code metrics an almost impossible task. On the contrary, process related metrics can be extracted quite accurately and easily from various DevOps tools such as GitHub, Jira, Jenkins, and Slack.

This paper aims to shed light on two major issues.The first issue is to identify, through the use of process metrics and extensive experimentation, the technique, or the combination of techniques and features, that best classify whether a software component (i.e. file) can be considered defect prone or not. The second issue is whether process metrics can be used instead of source code metrics and whether these can be used to train models that yield similar of better classification results in a single project or across projects. These issues are formalized by the following research questions:

RQ1: *By using a very large set of open source projects to experiment with, which is the best combination of classifiers which are fast, easily trainable and able to yield the best results as these are measured in terms of accuracy, precision, recall, F1, and AUC*?

RQ2: *What is an optimal subset of available process metrics which can be easily calculated and at the same time yield the best results when provided as input to different classifiers*?

RQ3: *Is it possible to perform defect-proneness classification using process metrics while maintaining classification performance measures comparable to similar techniques reported in the literature which use source code metrics*?

RQ4: *Is it possible to perform cross project defect proneness classification in the sense that data from different projects can be used to train a model which will then be used to perform defect proneness prediction on other unknown projects for which not enough training data may be available*?

For this paper we take an experimental approach, aiming to draw conclusions by applying the techniques under examination to a very large collection of open source projects. More specifically, we have considered a collection of 150 open source systems from which we have extracted various process metrics utilising a custom-made extraction tool. The open source projects were selected based on their complexity, size, prevalence, and the quality and availability of process repository data. The importance of the work reported in this paper lies on two parts. First, in the best of our knowledge, it is the first work which utilises such a large data set, providing thus a much more statistically significant result than previously reported works, and second providing answers to research questions which can assist researchers advance the state-of-the-art in the area.

The paper is organised as follows. Section II presents related work. Section III discusses the features and the feature extraction process. Section IV presents the different Machine Learning techniques which we have evaluated. Section V presents the results obtained, while Section VI discusses and interprets the obtained results. Finally, Section VII concludes the paper and offers pointers for future work.

II. Background

In the related literature there is a wealth of approaches for defect prediction using Machine Learning techniques. Two widely-used defect prediction techniques are regression and classification. The main purpose of regression techniques is to estimate the number of software defects on a software component. In contrast, classification techniques aim to tag whether a software module is faulty or not. It has been shown that classification models can be trained from defect data on earlier versions of the system being analyzed. Some of the most commonly used supervised learning techniques for defect prediction are outlined below.

Decision Trees (DT): Decision tree algorithms use tree structures to model decisions and their possible consequences. In decision trees each leaf node corresponds to a class label while attributes are represented as internal tree nodes.

Logistic Regression (LR): Logistic regression is a supervised classification algorithm whereby the target variable *O* (i.e output), can take on values in the interval $[0, 1]$ representing the probability for a given set of input features *I* to belong to class 1 or 0.

Random Forest (RF): RF is an ensemble type of learning method used for both classification and regression problems. The key idea behind RF is the construction of several decision trees at training time and outputting the mode/mean prediction of the individual trees.

Support Vector Machine (SVM): SVM is a discriminative classifier formally defined by a separating hyperplane. In SVMs, given a labeled training data set whereby each data item is marked as belonging to one or the other of two categories, the algorithm outputs an optimal hyperplane, which classifies new unseen data in one of these two categories.

k-Nearest Neighbors (k-NN): k-NN is a non-parametric method that can be used for both classification and regression problems. In both cases, the input consists of the k closest training examples in a feature space. The output depends on whether k-NN is used for classification or regression. In classification, the output is to categorize an input to one of equivalence classes. In regression, the output is to assign a value to the input, usually the average of the values of its closest k-neighbors.

Naive Bayes Classifiers (NB): These classifiers refer to a family of simple "probabilistic classifiers" based on applying Bayes' theorem and by considering a strong independence assumption between features, that is the presence or absence of a particular feature of a class is not related to the presence or absence of any other feature.

Neural Networks (NN): Neural Networks are nonlinear predictive structures that consist of interconnected processing elements called neurons that work together in parallel within a network to produce output, often simulating an unknown function or phenomenon.

Multi-layer Perceptron (MLP): MLPs refer to a class of feedforward artificial neural network (ANN). An MLP comprised of a directed graph of multiple layers of nodes which are fully connected to the nodes of the next layer. For training purposes, MLP utilizes a supervised learning technique defined as *backpropagation*.

Radial Basis Function (RBF) Networks: RBF Networks are a type of ANNs used to approximate through training the value of an unknown function. They are different from MLPs in the sense that they are feedforward networks comprising of only three layers, the input layer, the hidden layer and the output layer.

A. Defect Prediction using Machine Learning

A variety of machine learning methods have been proposed and assessed for addressing the software bug prediction problem. These methods include decision trees [4], neural networks [8], [12], Naive Bayes [7], [11], [17], support vector machines [6], Bayesian networks [16] and Random Forests [9].

1) Source CodeMetrics Approaches: Deciding whether a component has a high likelihood to be defective or not has been proved to have a strong correlation with a number of software metrics. Identifying and measuring software metrics is vital for various reasons, including estimating program

execution, measuring the effectiveness of software processes, estimating required efforts for processes, estimating the number of defects during software development as well as monitoring and controlling software project processes [21] [22]. Various software metrics have been commonly used for defect prediction, including lines of code (LOC) metrics, McCabe metrics, Halstead metrics, and object-oriented software metrics. Hence, the automated prediction of defective components from extracted software metrics evolved as a very active research area. [36]. In [14], Nagappan aims to find the best code metric to predict bugs. The conclusion of this work is that complexity metrics can successfully predict post-release defects, but there is no single set of metrics that is applicable to all systems. Hassan et. al have investigated the impact of different aspects of the modelling process to the end results and the interpretation of the models [28] [29] [30] [31] [40].

2) Process Metrics Approaches: In [10], Venkata et. al compared different machine learning models for identifying faulty software modules and they found that there is no particular learning technique that performs the best for all the data sets. In [5], Wang and Yao aim to find bugs without decreasing the overall performance of the model. In this process, they find that imbalanced distribution between classes in bug prediction is the root cause of its learning difficulty. Likewise, in our paper, we noted the issue and used re-sampling as described in detail in the section IV-C in an effort to minimise the impact of class imbalance to the quality of our results. Similarly, in [15], Zimmermann et. al propose an approach to predict bugs on cross-language systems. The work examined a large number of such systems and concluded that only 3.4% of the systems had precision and recall prediction levels above 75% . The authors also tested the influence of several factors on the success of cross-language prediction and concluded that there was no single factor which led to such successful predictions. The authors used decision trees to train the model and to estimate precision, recall, and accuracy before attempting a prediction across systems. Lastly, in [13], Hassan discusses how frequent source code "commits" in the repository negatively affect the quality of the software system, meaning that the more changes incurred to a file, the higher the chance that the file will contain critical errors. Furthermore, the author in [13] presents a model which can be used to quantify the overall system complexity using historical code-change data, instead of plain source code features.

III. Data Modeling

For the purposes of this study we have designed two separate data models. The first data model denotes the raw information which can be mined from software repositories, while the second data model denotes the post-processed raw data which are in a form that can be consumed by the machine learning algorithms we have experimented with.

The design goal of the first data model was to have a structure which would be easy to populate while maintaining a low memory profile, would facilitate data reconciliation of data entries originating from different devOps tools (e.g. GitHub, Jenkins, Jira), would be scalable, and would be able to support preprocessing workflows of varying complexity at high speeds. The schema for this data model is depicted in Fig. 1.

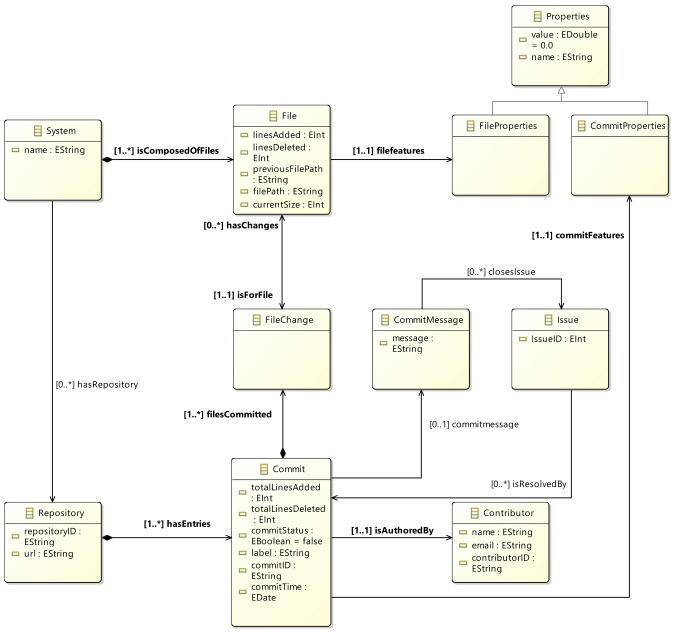

Fig. 1. Data Model for Raw Repository Data

The design of the second data model was to have a simple relational structure which can be easily imported as a tab or comma delimited file in various machine learning tools and which can be easily manipulated so that aggregate features can be easily computed. The features in this second data model are depicted in Table I.

A. Raw Data Model

For this study we have exhaustively collected process related metrics from 150 open source systems of various sizes and complexities. The list of the systems along with all the data obtained or computed are listed in the anonymous repository [23].[1] The profile of the data set we have considered is depicted in Table II. The data acquisition process is based on two steps. The first step is to utilize a custom made client-side extractor tool which is able to connect to and reconcile data obtained using various tools and namely GitHub, Bugzilla, Jira, and Jenkins. However, for this study we report results on data acquired only from the GitHub repositories of these 150 open source systems. The second step of the raw data acquisition process is to fuse the information extracted by each repository record into one repository which conforms to the raw data schema depicted in Fig. 1. The extractor application and its data fusion module is implemented using Python 3.

[1]Please note that the repository is anonymous for the time being, in order to protect the double blind review process and to facilitate the assessment of this work by the reviewers. Please do not distribute or use without prior arrangements with the authors.

As depicted in Fig. 1, the raw data model is founded on the concept of a *Commit*, the concept of a *File*, and the concept of a *CommitProperties*. The extracted information is represented as a Json file stored in a Mongo DB server. As such, a run-time model of a GitHub repository was created which held the information of the unique commit records. Every commit contained a list of *fileChanges* and the details for each of these files' change. This data model represents a GitHub record structure utilizing simple Python 3 objects which have a very low memory profile and initialization time.

In this data model a commit is uniquely identified by it's *commitID*, it contains attributes specific to it, including the *author*, the *commit-time*, *the files committed* as these are denoted by their *FileIds*, a *commit message*, the *overall lines added, deleted* as well as a tag field maintaining information about whether the *Commit*'s status is indicated as a *bug fixing* or as a *clean* Commit. For each *File* within a commit, the added, and deleted lines as well as the current size of the file are maintained. Storing the *fileID* is necessary since in case of a file changing locations the *fileID* remains the same even if the file changes name (the names are fully qualified names with respect to the root folder of a project). In addition to the version control model, another important component of this extractor system is the issue tracker component. This component is far simpler. It is a simple object maintaining a specific *issueID* together with the issue tag its message and its referenced *commitID*s. The data model is populated by initially downloading a complete repository from the corresponding GitHub site and then moving through each commit on the master branch adding the relative data iteratively in it, thus maintaining the initial structure. Once the model is populated it undergoes several steps of preprocessing. The first task is to remove all files that cannot contribute to a defect, such as any non-compilable and non-configuration related files. The next task is to use a simple heuristic to clean up the extracted commits so that only actual code changing commits remain. This entails removing all commits that are clearly annotated as a *refactoring* commit, and also removing all files which have been eventually removed from the system from all past commits. Finally all merge commits are also removed since they contain change information pertaining to different branches and will therefore introduce large ammount of noisy data points to the dataset. Given that this study is not focusing on defect introducing software changes, the removal of refactoring and merge commits from the dataset will not impact its ability to discern between faulty and healthy files. After the cleanup stage is completed, the most important remaining task is that of assigning the class label for each commit. This task is accomplished by parsing the commit message for terms that may indicate that it is a bug-fixing commit as opposed to a clean one, linking commits to issue tracker entries labeled as faults and optionally, applying the same parsing as above to the issue tracker messages. The heuristic terms used to tag a commit as a bug-fixing one are presented in Section IV-B below. This is an approach for automatically generating datasets for such applications which has been known to work in the related literature [26]. This approach however has the potential to produce a high number of false positives in the dataset because of the commit granularity level at which it is applied. This is due to the fact that some commits may be only partially defective leading to wrong labels for the non-defective part [39]. In this case the entirety of the commit and its modifications are annotated as bug-fixes, which is not acceptable for commits that contain a significant percentage of a systems' files.

TABLE I
FEATURES USED FOR SYSTEM TRAINING

F1: NoOfCommits (CF)	F2: LateNightCommits (LNC)
F3: TotalAddedLines (TAL)	F4: MaxAddedLines (MAL)
F5: AvgAddedLines (AAL)	F6: TotalDeletedLines (TDL)
F7: MaxDeletedLines (MDL)	F8: AvgDeletedLines (ADL)
F9: TotalChurn (TCF)	F10: MaxChurn (MC)
F11: AvgChurn (AC)	F12: TotalCoCommitSize (TCS)
F13: MaxCoCommitSize (MCS)	F14: AvgCoCommitSize (ACS)
F15: TotalDistinctAuthors (TDA)	F16: AgeInMonths (AIM)
F17: FractalValue (FRV)	F18: FailureIntensity (FI)

B. Post-Processed Data Model

Once the raw data are extracted they are post-processed in order to yield a data model suitable for input to various Machine Learning classification tools. Table I provides a list of the features considered for our study.

C. Explanation of the Features

The *FractalValue* [27] provides a measure for the contribution of different authors to a file. It can take any value in the range (0, 1] where a value of 1 means that a file has had a single author whereas a value close to 0 means that the file has had similar contributions from multiple different authors. *TotalCoCommitSize* for a file F_i in a system S is the count of all files $F_j \in S$ which have been committed alongside F_i, counting multiple occurrences of the same files.

IV. MACHINE LEARNING FRAMEWORK

In this section we discuss the machine learning algorithms used and technical details on how training and testing were conducted for this study.

A. Machine Learning Models Considered

For our experiments we have considered six different classification algorithms, and namely *(1)* Logistic Regression; *(2)* Support Vector Machines; *(3)* Multi-Layer Perceptron; *(4)* Decision Trees; *(5)* Random Forests and; *(6)* Naïve Bayes. The selection of these algorithms is based on the fact that these are the algorithms most commonly used in the related literature [38], [35], [36] [37].

Each of the aforementioned algorithms comes with its own benefits and drawbacks. The most important benefit was the speed at which these can be trained and evaluated while the most important drawback of all approaches except for Logistic Regression was the lack of explainability. Nevertheless, the combination of these algorithms is currently the de-facto standard in the related literature as base classifiers [36] [38].

B. Commit Tagging

As discussed in Section III-B the raw data repository is considered as a container of commits. Each commit is tagged in the repository as a *bug fixing commit* or a *clean commit*. This tagging is based *a)* on the label of the GitHub commit record itself, or in the absence of such a label by analyzing the *comments* section of the commit record. More specifically, if the *comments* section of the commit record contains any of the keywords *'bug', 'bugs', 'defect', 'defects', 'error', 'errors' 'fail', 'fails', 'failed', 'failing', 'failure', 'failures', 'fault', 'faults', 'fix', 'fixes', 'fixed', 'fixing', 'problem', 'problems', 'wrong'* which *may indicate* a bug fixing *intention*, then the commit is tagged as a *buggy commit* (label 1), otherwise as a *clean commit* (label 0).

C. File Tagging

In its turn, a commit is considered itself as a container of files. If a commit is tagged as a bug fixing one (see above), then all the files in the commit are also tagged as buggy. This is a heuristic that can introduce many false positives, but unfortunately in the absence of a gold standard this is the best approximation and it is also a heuristic which is used in most papers appearing in the related literature [26]. In our data model each file also contains details about the contribution of that file to the commit in terms of the lines of code added or deleted as a percentage of the overall number of lines of code added or deleted in the commit.

In the related research literature there is no authoritative set of tagged files which can serve as a gold standard. The only such data set is PROMISE which relates only to source code metrics and does not include process metrics. We have identified an intersection of 11 projects available in PROMISE [33] which have a tagging (buggy or clean) and for which we can extract process metrics. We have used these 11 projects to answer research question Q_3.

As most of the 150 systems which we have considered for this study are open source systems the operational life of which spans several years, we have split the commits into two eras. The rationale behind this split is that very old commits (e.g. commits which may be several years old) should not bear significant weight to the overall computation. The first era consists of the past 70% of the commits and the second era of most recent 30% of the commits. The experimentation set-up proceeds then as follows.

Feature Entries

Let $F_{i,j} =< m_1, m_2, \ldots, m_{18} >^{i,j}$ denote a feature value vector entry for file F_i participating in commit C_j, where m_k is the value of a feature f_k $k \in \{1, 2, ..18\}$ (see Table I) related to file F_i in commit C_j.

Let also $F_{i,S}$ = $< v_1, v_2, ..., v_{18} >^{i,S}$ be the feature value vector entry for file F_i across all commits C_j, $C_j \in S$, in which F_i appears in, and where each value v_p, $p \in \{1, 2, ..18\}$, is obtained by combining all correcponding m_p*'s* appearing in feature value vector entries $F_{i,j}$.

Let us also assume that the commits C_1 C_2, ... C_{j-1} = S_1 belong to the first era of 70% of commits and C_j C_{j+1}, ... C_n = S_2 belong to the era of the most recent 30% of system commits. Then the resulting feature value vector, for all commits $S = S_1 \cup S_2$ containing changes for this file F_i, will be $F_{i,S}$ = $F_{i,S_1 \cup S_2}$ will be $<< v_1, v_2, ..., v_{18} >^{i,S_1}, < v_1, v_2, ..., v_{18} >^{i,S_2} >$.

Tagging Process

If a commit C_k appearing in the most recent 30% of the system commits has been identified as a *bug fixing commit* then the metrics vector for $F_{i,k}$ is also tagged as buggy and is then denoted by the feature value vector $< m_1, m_2, \ldots, m_{18}, \mathbf{1} >^{i,j}$. The resulting feature vector accross all system commits S in which F_i participates is $<< v_1, v_2, ..., v_{18} >^{i,S_1}, < v'_1, v'_2, ..., v'_{18} >^{i,S_2}, \mathbf{1} >>$, where: S_1 is the set of all commits C_n F_i participates in and appearing in the past 70%, and S_2 is the set of all commits C_k F_i participates in, and are appearing in the most recent 30% of the system commits and at least one such C_k has been identified as a *bug fixing commit*.

If all commits C_k for a file F_i which appear in the last 30% have been tagged as a *clean commit* then the feature value vector $F_{i,k}$ is $< m_1, m_2, \ldots, m_{18}, \mathbf{0} >^{i,k}$, and the overall feature vector used is $<< v_1, v_2, ..., v_{18} >^{i,S_1}, < v'_1, v'_2, ..., v'_{18} >^{i,S_2}, 0 >>$ where: S_1 is the set of all commits C_n, F_i participates in, and are appearing in the past 70%, and S_2 is the set of all commits C_k F_i participates in, are appearing in the most recent 30% of the system commits and each such C_k has been identified as a *clean commit*.

If for a file F_i all of its commits C_n appear in the past 70% of the system commits (i.e. in the first era), then the feature value vector, is $< v_1, v_2, ..., v_n, DC >^{i,S_1}, < NULL, NULL, ...NULL >^{i,\emptyset} >>$ (where DC stands for "Don't care" value) and where: S_1 is the set of all commits C_n, F_i participates in, and are appearing in the past 70%.

Example

As an example, consider the file F_{10} which appears in commits C_5, C_{50}, and C_{1000} where C_5, C_{50} belong to the first era (past 70%) while C_{1000} belongs to the recent 30% and is tagged as a bug fixing commit. Then the file F_{10} will have the following feature vector:

$<< v_1, v_2, \ldots, v_{18}, >^{10,\{5,50\}},$
$< v_1, v_2, ..., v_1 8, 1 >^{10,\{1000\}} >>$

and which is produced by aggregating the feature value vectors:

$< m_1, m_2, ..., m_{18}, DC >^{10,5}$
$< m'_1, m'_2, \ldots, m'_{18}, DC >^{10,50}$
$< m''_1, m''_2, \ldots, m''_{18}, 1 >^{10,1000}$

The post processed data model is essentially a relational table where each line in the table is the feature vector entry of the file $< v_1, v_2, \ldots, v_{18}, Tag >^{i,S}$.

The obtained results are then the data considered on this study in order to answer questions $Q_1 - Q_4$.

Rebalancing

In the related research literature rebalancing is applied in order to avoid overfitting classifiers or other *ML* models to the majority class. Likewise we used the *random oversampling* technique [2] to tackle this threat, accepting that it may contribute to drift bias in the generated models [29]. The technique involves randomly selecting instances from the minority class with replacement before the training stage, to create a new set of the minority class' instances which will resemble in cardinality that of the majority class.

D. Training and Test Set Data and Bias

Once we have calculated the feature vector entries for each file, we follow the 80-20 split rule for testing and training, and we consider all the entities in the post processed data (i.e. entries from the fisrt and second era of the commits). This was done so that the produced results will be comparable to the studies published in the relevant research literature [18]. Aiming at reducing the effect of outliers across all 150 systems considered, we trained and applied a scaler on the training data to decrease the effective range of all feature values, and also applied rebalancing on the minority and majority classes as explained in Section IV-C.

E. Evaluation and Performance metrics

The norm for extracting meaningful results from *ML* models is the use of the stratified k-fold cross validation technique [18]. The bootstrap and leave-one-out validation techniques were also used to provide a better understanding of how the models would perform during the application of the trained model. The performance metrics used in this study are the ones most frequently mentioned in the literature [29] [32]. In total, 7 different performance measures were calculated for each one of the variations of the technique and namely *precision*, *recall*, *accuracy*, *F1-score*, *Brier-score*, *Receiver-operator-characteristic/area-under-curve*, and where possible, *support*. In the context of academic research it has been argued that an overall high *F1-score* as well as a high *area-under-curve* are good indicators to identify whether a classifier is a successful one or not. Given, however, that F1 is a combination of Precision and Recall, it means that a satisfactory value for it is not necessarily the result of an optimal combination of it's constituent values. However, in industry, it is often preferred to maintain high precision even at the expense of recall. The rationale is that investigating false positives may require significant effort, or may result to the prediction system not being easily adopted by developers.

V. Evaluation Studies

In this section we present the details of the studies we have conducted for answering the research questions Q_1 - Q_4. As stated, some basic statistics of the 150 open source projects considered, are depicted in Table II.

TABLE II
Project Statistics

Project Size (in files)	Avg. LOC	No. of Projects	Avg. Age (in years)	Avg. No. of Commits
148,740 – 20,000	15 MLOC	4	13	77,230
19,999 – 10,000	2.3 MLOC	4	11	18,870
9,999 – 5,000	1.3 MLOC	5	14	13,400
4,999 – 2,000	500 KLOC	18	10	7,288
1,999 – 1,000	457 KLOC	12	15	13,906
999 – 500	231 KLOC	26	10	3,900
499 – 200	86 KLOC	32	9	2,213
199 – 12	50 KLOC	47	9	1,325

A. Study 1: Identification of Best Classifiers

The first study aims to identify through experimentation the combination of the best classifiers to be used for defect prediction (classification). Our study here is by far not the first study of its kind. In [36] the authors have reviewed various research approaches and concluded that the best results are consistently given by the application of simple modelling techniques such as Logistic Regression and Naive Bayes. Similarly, the authors in [34] reported that the best features to use are *owner experience, overall developer experience, owner contributed lines, minor contributor count, and distinct dev count* of which only the last one is used in this approach. However, to our knowledge the study in this paper is the first of its kind in that it uses *a)* a very large, comprehensive, and statistically significant sample of 150 systems in a quest to obtain conclusive results in the topic, and *b)* repository process metrics as opposed to source code metrics which as mentioned above do not require the use of specialised parsers and source code metrics calculators.

1) Study Set-up: For this study we have obtained feature vector entries collected from post-processing raw data extracted from 150 GitHub [1] repositories, as discussed in Section III-A and Section III-B. The Data were split into k-folds to apply k-fold cross validation were k=5. For each of the folds the training data had to be rebalanced as most projects exhibit uneven numbers of fault-prone over healthy files. The rebalancing was performed by oversampling the minority class. The data were also normalized in order to map feature values to be in the range of 0 and 1. This was done by utilising a scaler utility available in Python while the testing data were scaled using the same scaler model. Finally the training data were used for training and the remaining for testing over 5 folds and the results extracted depict the average of the score extracted for each fold.

2) Obtained Results: Due to space limitations, we only report the highlights of the obtained results. The full results list can be found on the data repository accompanying this paper [23]. For this study we have used all the combinations of the following classifiers: Decision Trees (DT), Random Forests (RF), Linear Regression (LR), Multi-Layer Perceptron (MLP), Support Vector Machines, and Gaussian Naive Bayes

TABLE III
RESULTS OF CLASSIFIER COMBINATIONS

Classifier Combination	Accuracy Range Median Accuracy Avg. Accuracy	AUC Range Median AUC Avg. AUC	F1 Range Median F1 Avg. F1
DT_LR_RF	0.7 - 1.0 0.89 0.89	0.61 - 1.0 0.85 0.85	0.0 - 1.0 0.81 0.75
DT_GNB_LR_MLP_RF	0.69 - 1.0 0.88 0.88	0.64 - 1.0 0.85 0.86	0.0 - 1.0 0.79 0.75
DT_GNB_MLP_RF_SVM	0.61 - 1.0 0.87 0.88	0.64 - 1.0 0.85 0.85	0.0 - 1.0 0.79 0.75
DT_GNB_LR_MLP_RF_SVM	0.61 - 1.0 0.87 0.87	0.61 - 1.0 0.84 0.85	0.0 - 1.0 0.77 0.74
DT_GNB_LR_RF_SVM	0.65 - 1.0 0.87 0.87	0.62 - 1.0 0.84 0.85	0.0 - 1.0 0.77 0.74
LR_MLP_RF	0.61 - 1.0 0.87 0.87	0.63 - 1.0 0.85 0.86	0.0 - 1.0 0.79 0.75
DT_LR_MLP_RF_SVM	0.65 - 1.0 0.87 0.87	0.65 - 1.0 0.86 0.86	0.0 - 1.0 0.79 0.75
DT_LR_MLP	0.57 - 1.0 0.87 0.87	0.62 - 1.0 0.86 0.86	0.0 - 1.0 0.79 0.75
LR_MLP_RF_SVM	0.53 - 1.0 0.86 0.87	0.6 - 1.0 0.85 0.86	0.0 - 1.0 0.78 0.73
MLP_RF_SVM	0.65 - 1.0 0.86 0.87	0.65 - 1.0 0.85 0.86	0.1 - 1.0 0.79 0.74
DT_LR_MLP_SVM	0.57 - 1.0 0.86 0.87	0.62 - 1.0 0.85 0.85	0.0 - 1.0 0.77 0.73
GNB_LR_MLP_RF_SVM	0.65 - 1.0 0.86 0.87	0.64 - 1.0 0.85 0.86	0.0 - 1.0 0.78 0.74
DT_GNB_LR_MLP_SVM	0.65 - 1.0 0.86 0.87	0.6 - 1.0 0.85 0.86	0.0 - 1.0 0.78 0.74
DT_MLP_SVM	0.61 - 1.0 0.86 0.87	0.65 - 1.0 0.86 0.86	0.0 - 1.0 0.79 0.74
LR_RF_SVM	0.65 - 1.0 0.86 0.86	0.65 - 1.0 0.84 0.86	0.0 - 1.0 0.77 0.74
MLP	0.61 - 1.0 0.85 0.86	0.64 - 1.0 0.85 0.86	0.1 - 1.0 0.78 0.74
DT_LR_SVM	0.53 - 1.0 0.86 0.86	0.6 - 1.0 0.85 0.86	0.0 - 1.0 0.77 0.73
GNB_LR_MLP	0.61 - 1.0 0.86 0.86	0.59 - 1.0 0.84 0.85	0.04 - 1.0 0.77 0.73
GNB_MLP_SVM	0.57 - 1.0 0.85 0.86	0.63 - 1.0 0.85 0.86	0.0 - 1.0 0.77 0.73
LR_MLP_SVM	0.61 - 1.0 0.85 0.86	0.63 - 1.0 0.85 0.86	0.1 - 1.0 0.77 0.74

(GNB). These combinations in all 150 projects produced 9,324 individual results. The summary of these results are depicted in Table III.

The results indicate that if the Accuracy criterion is to be considered first amongst the top ranked combinations with respect to AUC, and F1 score, then the best combination of classifiers across all projects are the DT_LR_RF, followed by the DT_GNB_LR_MLP_RF. Furthermore, this observation becomes more pronounced for projects for which the Buggy/Clean file ratio is more that 0.5 (see Section VI). The best combination of classifiers as grouped by the ratio Buggy/Clean files is depicted in Table IV.

TABLE IV
AUC FOR THE 10 BEST CLASSIFIER COMBINATIONS COMPARED TO BUGGY/CLEAN RATIO RANGE

Classifier	Avg. F1 and avg. AUC for B/C ¡ 0.25	Avg. F1 and avg. AUC for B/C = [0.25 - 0.5)	Avg. F1 and avg. AUC for B/C $\geq$ 0.51
DT_LR_RF	0.55 0.87	0.74 0.85	0.84 0.85
DT_GNB_LR_MLP_RF	0.55 0.87	0.74 0.85	0.84 0.85
DT_GNB_MLP_RF_SVM	0.55 0.87	0.74 0.85	0.84 0.85
DT_GNB_LR_MLP_RF_SVM	0.55 0.87	0.74 0.85	0.84 0.85
DT_GNB_LR_RF_SVM	0.55 0.87	0.74 0.85	0.84 0.85
LR_MLP_RF	0.55 0.87	0.74 0.85	0.84 0.85
DT_LR_MLP_RF_SVM	0.55 0.87	0.74 0.85	0.84 0.85
DT_LR_MLP	0.55 0.87	0.74 0.85	0.84 0.85
LR_MLP_RF_SVM	0.55 0.87	0.74 0.85	0.84 0.85
MLP_RF_SVM	0.55 0.87	0.74 0.85	0.84 0.85

B. Study 2: Identification of Best Features

This part of the study aims to identify the optimal combination of features that can be used for creating defect proneness classification models. This kind of research has been carried out in past work [34], where the authors have proposed and used different combinations of process metrics features and investigated their significance, stability, staticness and portability as well as their performance for defect prediction.

1) Study Set-up: For this study the classifier used was the DT_LR_RF which was identified as optimal as shown in Table III, and was applied on all possible combinations of the features. For the evaluation of the trained models the k-fold cross validation with k=5 was used. The extracted results represent the average over the 5 folds. Rebalancing was used to equalise the instances pertaining to the minority and majority classes, and scaling applied to normalise the data points as to facilitate a better fitting of the models to the data prior to training. The testing data were not rebalanced but the same scaling was applied to them as well.

2) Obtained Results: Due to space limitations we report the summary of the obtained results for all projects, per feature combination in Table V. For this study we have used all combinations of the following Features: Number of Commits Feature(CF), Total Distinct Authors (TDA), Total Churn Feature (TCF), and Total CoCommits Size (TCS). These combinations in all 150 projects produced 2,220 individual

TABLE V
RESULTS OF FEATURE COMBINATIONS

Feature Combination	a. Accuracy Range b.Median Accuracy c. Average Accuracy	a. AUC Range b. Median AUC c. Average AIC	a. F1 range b. Median F1 c. Average F1
CF_TCF_TCS_TDA	0.68 - 1.0 0.89 0.89	0.58 - 1.0 0.86 0.86	0.0 - 1.0 0.8 0.76
CF_TCF_TCS	0.7 - 1.0 0.89 0.89	0.55 - 1.0 0.85 0.85	0.0 - 1.0 0.81 0.75
CF_TCS_TDA	0.63 - 1.0 0.88 0.88	0.59 - 1.0 0.85 0.85	0.0 - 1.0 0.8 0.76
CF_TCS	0.66 - 1.0 0.88 0.88	0.6 - 1.0 0.85 0.85	0.0 - 1.0 0.81 0.76
TCF_TCS_TDA	0.72 - 1.0 0.88 0.88	0.64 - 1.0 0.84 0.84	0.0 - 1.0 0.8 0.75
TCF_TCS	0.7 - 1.0 0.87 0.87	0.55 - 1.0 0.82 0.83	0.0 - 1.0 0.78 0.73
TCS_TDA	0.6 - 1.0 0.88 0.87	0.46 - 1.0 0.83 0.84	0.0 - 1.0 0.79 0.74
CF_TCF_TDA	0.73 - 1.0 0.86 0.87	0.6 - 1.0 0.83 0.83	0.0 - 1.0 0.78 0.72
CF_TCF	0.62 - 1.0 0.85 0.86	0.5 - 1.0 0.81 0.82	0.0 - 1.0 0.76 0.71
TCF_TDA	0.69 - 1.0 0.85 0.85	0.61 - 1.0 0.8 0.81	0.0 - 1.0 0.75 0.7
CF_TDA	0.5 - 1.0 0.84 0.85	0.6 - 1.0 0.81 0.82	0.13 - 1.0 0.77 0.71
CF	0.33 - 1.0 0.82 0.84	0.4 - 1.0 0.8 0.81	0.13 - 1.0 0.75 0.69
TCS	0.47 - 1.0 0.84 0.83	0.44 - 1.0 0.79 0.79	0.0 - 1.0 0.71 0.67
TCF	0.63 - 1.0 0.82 0.83	0.5 - 1.0 0.76 0.77	0.0 - 1.0 0.69 0.65
TDA	0.39 - 1.0 0.8 0.81	0.4 - 1.0 0.81 0.81	0.07 - 1.0 0.73 0.68

results. The full results list can be found on the data repository accompanying this paper [23].

C. Study 3: Comparison of Process and Source Code Metrics

This study aims at performing a comparison between the efficiency of using Source Code Metrics versus using Process Metrics for carrying out Fault-Proneness prediction. It is carried out on a subset of data made available for software engineering research as part of the PROMISE repository [33].

1) Study Set-up: All combinations of the available classifiers were used for this experiment and the Feature combination used was the one identified as optimal in Section V-B. To select the subset of systems on which to conduct this study we manually investigated the contents of the PROMISE repository to identify projects for which a valid Git repository is still available. Given the age of the repository and it's specific structure this process yielded only 11 systems for which process metrics could be mined. For these 11 systems the data between the PROMISE dataset and our system were reconciled. The reconciliation process consisted of only using data pertaining to files present both in data extracted from Git repositories and in the PROMISE dataset. In addition, the files had to be active with at least a single commit in the latest 30% of the total commits of the system. The files' classes were set from the manually curated PROMISE dataset. The process used afterwards is the same as in Sections V-A and V-B. The data were rebalanced in both cases and independently scaled. The process presented so far was designed to give both approaches an equal ammount of data and to be easily replicated by other researchers. The evaluation of the models was implemented using k-fold cross validation where k=5 for each of the approaches, Source-Code Metrics and Process Metrics respectively, and the presented results depict the average over all 5-folds of this evaluation.

2) Obtained Results: Due to space limitations, we report here the highlights of the obtained results. The full results list can be found on the data repository accompanying this paper [23]. For this study a voting classifier was utilised using all combinations of the following classifiers: Decision Trees (DT), Random Forests (RF), Linear Regression (LR), Multi-Layer Perceptron (MLP), Support Vector Machines(SVM), and Gaussian Naive Bayes(GNB). The Features used were: Number of Commits Feature(CF), Total Distinct Authors (TDA), Total Churn (TC), and Total CoCommitsSize Feature (TCS) for extracting process metrics and all features available were used from the PROMISE dataset. This process was applied on 11 projects and yielded a total of 756 results for each metric type. The results are shown in Table VI.

D. Study 4: Cross Project Validation

For this study, we have trained the classifiers in a collection of projects (training set) and we have applied them to another set projects (testing set) for comparing the obtained results with the ones obtained when the classifiers are trained and applied only in one project.

1) Study Set-up: For this study the optimal classifier identified in Section V-A and the optimal feature combination identified in Section V-B were used. To prepare the data we filtered the available systems selecting only those having less than 20K and more than 250 files and then split these into three performance classes using the ratio of fault-prone over healthy files in the system. This yielded a total of 26 projects with a B/C ratio in the interval [0, 0.25), 20 projects with a ratio in the interval [0.25, 0.5) and 44 projects with a B/C ratio $\geq$ 0.5 (see also Table IV). These groups were then divided into two randomly selected groups, and one group was used for training a model while the other group was used for evaluation. Given the uneven size of the different systems it was necessary to upsample the data available for each one of them so as to have all projects represented approximately equally in the training set and avoiding it being dominated by the largest systems. Rebalancing of the minority and majority classes was

TABLE VI
PROCESS METRICS VS SOURCE CODE METRICS

Classifier Combination	Process Metrics Avg. F1 Median F1 Avg. AUC Median AUC	Source Code Metrics Avg. F1 Median F1 Avg. AUC Median AUC
DT_GNB_RF	0.8 0.93 0.77 0.74	0.58 0.58 0.67 0.63
DT_MLP_RF	0.81 0.93 0.77 0.76	0.6 0.61 0.67 0.65
DT_RF_SVM	0.81 0.93 0.75 0.74	0.6 0.62 0.67 0.65
RF	0.8 0.93 0.79 0.76	0.59 0.61 0.66 0.63
DT_LR_RF	0.81 0.92 0.76 0.77	0.61 0.61 0.68 0.65
DT_GNB_MLP_RF_SVM	0.8 0.84 0.75 0.73	0.59 0.59 0.68 0.65
DT_GNB_LR_MLP_SVM	0.79 0.82 0.75 0.75	0.59 0.59 0.68 0.67
GNB_LR_MLP_RF_SVM	0.79 0.82 0.76 0.74	0.58 0.6 0.68 0.66
DT_LR_SVM	0.79 0.82 0.75 0.74	0.58 0.6 0.65 0.65
SVM	0.79 0.82 0.76 0.75	0.57 0.59 0.65 0.64

carried out on the upsampled data separately for each project to provide the training algorithm with equal ammounts of positive and negative instances. In this study there was no reason to use the k-fold cross validation technique as the evaluation of the trained model happened on other systems than the ones used for training.

2) Obtained Results: For this study a total of 45 results were obtained one for each system used for testing. Due to space limitations we report here a summary of these results. The full set of results can be found in [23]. The results are presented in a condensed form and grouped by faulty over healthy ratio in Table VII.

VI. DISCUSSION

A. General Observations

The first observation is that each software system is unique, and there is no single best classifier which can be used to provide accurate defect proneness classification results for all projects. What we have observed is that a classifier or a collection of classifiers can produce high performance scores (i.e. high accuracy, high F1, and high AUC) for one project, and poor scores on another (see also "No Free Lunch Theorem" [3]).

TABLE VII
CROSS PROJECT WITH REBALANCING

	Cross vs. Own	Cross vs. Own	Cross vs. Own
B/C Value	Avg. Accuracy Accuracy Median	Avg. AUC AUC Median	Avg. F1 F1 Median
B/C < 0.25	0.75 vs. 0.60 0.81 vs. 0.63	0.77 vs. 0.94 0.78 vs. 0.94	0.4 vs. 0.63 0.43 vs. 0.64
B/C $\in$ [0.25,0.5)	0.69 vs. 0.76 0.74 vs. 0.76	0.73 vs. 0.86 0.76 vs. 0.86	0.62 vs. 0.78 0.66 vs. 0.77
$B/C \geq 0.5$	0.73 vs. 0.86 0.74 vs. 0.84	0.75 vs. 0.88 0.74 vs. 0.85	0.76 vs. 0.85 0.77 vs 0.84

The second observation is that we were not able to identify a feature, or a collection of features, that guarantee (i.e. with certainty) that such a classifier or a set of classifiers can be trained to always yield high performance scores. The only measure we have found to be a very good indicator of quality results is the Buggy/Clean ratio, where in the vast majority of cases, if a project has a Buggy/Clean ratio $\geq$ 0.5 it is almost certain that there exist a classifier or a combination of classifiers which can produce F1 and AUC scores higher that 0.85. This essentially means that classifiers work best when the data (number of buggy vs. clean files) are balanced. Classifiers on projects with Buggy/Clean ratio values less than 0.2 almost certainly perform poorly.

The third general observation is that there is a need to devise techniques for accurate tagging (i.e. to assign a *buggy* or *clean* label to a file) to be used for training purposes. The heuristics used so far in the literature and in this study, may introduce many false positives for training, skewing thus the obtained results. Furthermore, there is a need to devise techniques for introducing a temporal effect on the data, meaning that distant past commits should carry less weight than recent commits.

Overall, *ML* shows to be an interesting technique for defect proneness classification but we have not reached a point yet to identify how these classifiers can be trained effectively to yield trustworthy results for an arbitrary given project.

B. Detailed Observations

For research question RQ_1: The combinations of classifiers which included one or more tree-based models performed on average better than any other combination of classifiers. This may have to do with the nature of the problem which lends itself more naturally to a binary type of classification (i.e. *Buggy* or *Clean*) in which tree-based classifiers may perform better. We can say that combinations of classifiers which include Decision Trees and Random Forests, on average, outperform other combinations. Having said that, our observation is that there are no guarantees that these classifiers will perform as well when applied to a new arbitrary project, but there is a higher likelihood they will.

For research question RQ_2: By examining the obtained results the features having the highest probability of generating

high quality results is the combination of all four features, followed by the combination of CF (number of commits) and TCS (total co-commit size) and TCF (total churn). However, looking at the minimal set of features which can be used and still produce high quality results is any combination of two of CF (number of commits) and TCS (total co-commit size) and TCF (total churn). Another observation is that the TDA (total distinct authors) feature on its own does not provide high quality results, but only when combined with other features.

For research question RQ_3: Here we can say that process metrics seem to outperform the source code metrics for defect-proneness classification purposes. This is an encouraging observation as process metrics are language agnostic and their compilation does not require specialised parsers and metrics extractors. This result was anticipated, as software metrics may more often exhibit a variability in values over the different commits while the files maintain their tag value (i.e. Buggy or Clean), and vice versa, that is metrics may more often exhibit a variability in values over the different commits while the files change their tag value. This property may generate conflicting data for the classifier. In contrast, process metrics may exhibit a variability too, but maintain a better type of "history" feature values as the project evolves.

For research question RQ_4: Here our observation is that cross project classifier training and application does not yield high quality results. When classifiers are trained in some projects and applied to other projects the classification results are not as accurate as when the classifier is applied to the same project as the one it is trained on. This may be explained from the fact that the profiles of process metric values are kind of project specific as they depict the history of the project and the activity on each file. This is an interesting result, as it disputes the case of one trained model fits all.

C. Threats to Validity

We identify three threats. The first threat has to do with the way tagging is performed in order to create a training set. For our study we consider for tagging purposes the most recent 30% of the commits, while we maintain historical information from the past 70% of commits. For the feature value vectors of the distant past 70% of the commits we are either providing a DC value or no value (e.g. see Section IV-C). This creates the potential for false positives to be generated during tagging. The second threat has to do with how files are tagged within a single commit. For this study, if a file participates on a bug fixing commit then we consider all files in the commit as buggy. This is an overestimate and introduces the possibility of false positives. The most accurate approach would be to be able to tag all files in all commits in the history of the project with their correct label. However, this would be almost impossible for such large projects we have considered in the course of this study. It would be though a valid approach for new projects, where accurate labeling can commence on early stages of the project. The third threat has to do with the used features. Since the purpose of the study was to provide results from a large data set and not to propose new features, we have considered features which are commonly used in the research literature. There may be other features which relate to process metrics and which the community has not considered yet, which may produce good defect proneness classification results. This can also be considered an open problem for further investigation.

VII. Conclusion

This paper reports on the results of a set of experiments conducted in order to evaluate the use of Machine Learning for defect proneness classification. Over the past few years we have seen a tremendous growth on research and publications related to Machine Learning for software maintenance, and in particular for defect proneness classification and defect prediction. Even though there is a significant body of work conducted on evaluating Machine Learning techniques for defect proneness classification, the significance of this paper is that it is the first work to our knowledge that examines such a large corpus of open source data aiming to concretely address four key research questions which relate to experimentally identifying the best classifiers, the best features, whether process metrics outperform software metrics as predictors, and whether cross project training and application can be a viable option with respect to the quality of the obtained results. The experiments conducted revealed a number of observations. First, Machine Learning techniques are not guaranteed to perform well in all projects. Each software project has a specific life-cycle and "personality" profile of its own, and "a one classifier fits all" approach is not feasible. Second, we have seen that Machine Learning techniques are more likely to perform well when the buggy/clean ratio of the system files is between 0.5 and 2. Note that a ratio of 1 indicates that there are as many buggy files and clean files. Third, there was a clear indication that process metrics perform better or, in some cases, at least as well as software metrics. This implies that there is a strong indication that process related metrics can safely be used as predictors. Fourth, training the classifiers in one set of systems and applying on another is not a good approach, as the best classification results are obtained when the classifier is applied on the same system it is trained on. Overall, *ML* for defect proneness classification is a promising area of work that still has a number open problems to investigate. These include identifying new features to use as predictors, creating better tagging tools, and combining *ML* with static and dynamic analysis to increase the performance of the classifiers.

References

[1] Github, Build software better, together - https://github.com.

[2] Guillaume Lemaître, Fernando Nogueira, and Christos K Aridas. 2017. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *The Journal of Machine Learning Research* 18, 1 (2017), 559–563.

[3] Wolpert D.H., Macready W.G. . "No Free Lunch Theorems for Optimization" . IEEE Transactions on Evolutionary Computation, Vol. 1, pp. 67-82, 1997.

[4] Taghi M Khoshgoftaar and Naeem Seliya. 2002. Tree-based software quality estimation models for fault prediction. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*. IEEE, 203–214.

[5] Shuo Wang and Xin Yao. 2013. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability* 62, 2 (2013), 434–443.

[6] David Gray, et. al, 2009. Using the support vector machine as a classification method for software defect prediction with static code metrics. In *International Conference on Engineering Applications of Neural Networks*. Springer, 223–234.

[7] Burak Turhan and Ayse Bener. 2009. Analysis of Naive Bayes' assumptions on software fault data: An empirical study. *Data & Knowledge Engineering* 68, 2 (2009), 278–290.

[8] Mie Mie Thet Thwin and Tong-Seng Quah. 2005. Application of neural networks for software quality prediction using object-oriented metrics. *Journal of systems and software* 76, 2 (2005), 147–156.

[9] Cagatay Catal and Banu Diri. 2009a. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences* 179, 8 (2009), 1040–1058.

[10] Venkata Udaya B Challagulla, Farokh B Bastani, I-Ling Yen, and Raymond A Paul. 2008. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools* 17, 02 (2008), 389–400.

[11] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data mining static code attributes to learn defect predictors. *IEEE Trans. on Software Engineering* 1 (2007), 2–13.

[12] Jun Zheng. 2010. Cost-sensitive boosting neural networks for software defect prediction. *Expert Systems with Applications* 37, 6 (2010), 4537–4543.

[13] Ahmed E Hassan. 2009. Predicting faults using the complexity of code changes. In *Proceedings of the 31st Intl. Conf. on Software Engineering*. IEEE Comp. Society, 78–88.

[14] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*. ACM, 452–461.

[15] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 91–100.

[16] Ahmet Okutan and Olcay Taner Yıldız. 2014. Software defect prediction using Bayesian networks. *Empirical Software Engineering* 19, 1 (2014), 154–181.

[17] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. 2015. An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology* 59 (2015), 170–190.

[18] T. T. Wong, "Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation", Pattern Recognition, 48(9): pp.2839-2846, 2015.

[19] Shyam R. Chidamber and Chris F. Kemerer, "A metrics suite for object oriented design". IEEE Trans. Software Eng., 20(6):476 pp. 493, 1994.

[20] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object oriented Design Metrics as Quality Indicators," Trans. on Software Eng. , 22(10):pp. 751 - 761, 1996.

[21] D. Romano, M. Pinzger, "Using Source Code Metrics to Predict Change-Prone Java Interfaces", 27th International Conference on Software Maintenance, 2011.

[22] F. Toure, M. Badri, L. Lamontagne, "Predicting different levels of the unit testing effort of classes using source code metrics: a multiple case study on open-source software", Innovations in Systems and Software Engineering 14, pp. 15-46, 2018.

[23] https://figshare.com/s/b3741f9d0c3a939669db

[24] F. Zhang, F. Khomh, Y. Zou and A. E. Hassan, "An Empirical Study on Factors Impacting Bug Fixing Time," 2012 19th Working Conference on Reverse Engineering, pp. 225-234, 2012.

[25] Menzies, T., Milton, Z., Turhan, B. et al. "Defect prediction from static code features: current results, limitations, new approaches". Automated Software Eng. 17, pp. 375–407, 2010.

[26] Z. Toth, P. Gyimesiand R. Ferenc. "A Public Bug Database of GitHub Projects and Its Application in Bug Prediction". Springer, Intern.Conf. on Computational Sciences and Its Applications pp. 625 - 638, 2016.

[27] A. Tornhill. "Your code as a crime scene: use forensic techniques to arrest defects, bottlenecks, and bad design in your programs." Pragmatic Bookshelf, 2015.

[28] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto. "The Impact of Automated Parameter Optimization on Defect Prediction Models". IEEE Trans. Software Eng., 45(7):pp. 683 - 711, 2018.

[29] C. Tantithamthavorn, A. E. Hassan, K. Matsumoto. "The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models". IEEE Trans. Software Eng., DOI 10.1109/TSE.2018.2876537, IEEE, 2018.

[30] M. Kondo, C.P. Bezemer, Y. Kamei, A. E. Hassan, and O. Mizuno. "The Impact of feature reduction techniques on defect prediction models." Empirical Software Engineering 24:pp. 1925 - 1963, 2019.

[31] J. Jiarpakdee, C. Tantithamthavorn, A. E. Hassan. "The Impact of Correlated Metrics on the Interpretation of Defect Prediction Models." IEEE Trans. Software Eng. early access, DOI 10.1109/TSE.2019.2891758, 2019.

[32] Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance", IEEE Trans. Software Eng., 39 (6): pp. 757-773, 2013.

[33] J. Sayyad Shirabad and T.J. Menzies, "The PROMISE Repository of Software Engineering Databases", 2005.

[34] F. Rahman, P. Devanbu, "How, and Why, Process Metrics are Better," in Proceedings of the International Conference on Software Engineering, 2013, pp. 432 – 441.

[35] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings". Trans. Software Eng., 34(4): pp. 485 - 496, 2008.

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

[37] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison". Empirical Software Eng. 17, pp. 531–577, 2012.

[38] C. Tantithamthavorn and A. E. Hassan, "An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges". in Proceedings of the International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 286 – 295, 2018.

[39] L. Pascarella, F. Palomba, A. Bacchelli, "Fine-grained just-in-time defect prediction", Journal of Systems and Software 150: pp.22-36, 2018.

[40] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto, "The Impact of Automated Parameter Optimization on Defect Prediction Models". IEEE Trans. on Software Eng. 45(7): pp. 683 - 711, 2019.