Looking for Software Defects? First Find the Nonconformists

Sara Moshtari, Joanna C.S. Santos, Mehdi Mirakhorli, Ahmet Okutan
Rochester Institute of Technology
Rochester, United States
sm2481@rit.edu, jds5109@rit.edu, mxmvse@rit.edu, Axoeec@rit.edu

***Abstract*—Software defect prediction models play a key role to increase the quality and reliability of software systems. Because, they are used to identify defect prone source code components and assist testing activities during the development life cycle. Prior research used supervised and unsupervised Machine Learning models for software defect prediction. Supervised defect prediction models require labeled data, however it might be time consuming and expensive to obtain labeled data that has the desired quality and volume. The unsupervised defect prediction models usually use clustering techniques to relax the labeled data requirement, however labeling detected clusters as defective is a challenging task. The Pareto principle states that a small number of modules contain most of the defects. Getting inspired from the Pareto principle, this work proposes a novel, unsupervised learning approach that is based on outlier detection. We hypothesize that defect prone software components have different characteristics when compared to others and can be considered as outliers, therefore outlier detection techniques can be used to identify them. The experiment results on 16 software projects from two publicly available datasets (PROMISE and GitHub) indicate that the k-Nearest Neighbor (KNN) outlier detection method can be used to identify the majority of software defects. It could detect 94% of expected defects at best case and more than 63% of the defects in 75% of the projects. We compare our approach with the state-of-the-art supervised and unsupervised defect prediction approaches. The results of rigorous empirical evaluations indicate that the proposed approach outperforms existing unsupervised models and achieves comparable results with the leading supervised techniques that rely on complex training and tuning algorithms.**

***Index Terms*—Defect prediction, unsupervised learning, outlier detection, software metrics, software quality**

I. Introduction

There have been decades of research about defect prediction with the goal of building models that can help development teams to detect defect-prone components in software projects [1]–[26]. Defect prediction models use a variety of metrics including source code and software process metrics, as well as data mining techniques to identify defect-prone source code components. Most of the prior software defect prediction studies have used supervised classification methods [27]. In the supervised approach, a model is trained with labeled instances, and then used to predict the label of the unseen instances. A key challenge of the supervised defect prediction models is that they require training data which might be time consuming and expensive to obtain. Many supervised models proposed so far [1], [3], [4], [6], [7], [11] use data from previous releases of a project to predict faulty components in its next releases (*within-project defect prediction*). However, in real life, software projects may not always have historical data that can be used to build predictive models [28]. Therefore, *Cross-project defect prediction models* have been proposed to address the requirement of training data for each project [8]–[17], [29]. These approaches use data from one or more projects to predict faulty components in other projects. However, these methods do not always work well, because of the heterogeneity of the software projects and obtaining training data with desired quality is still a challenging task for cross-project defect prediction methods [28]. Some other works [18]–[20], [27], [28], [30] propose to use *unsupervised* learning approaches for software defect prediction. Some of the unsupervised approaches which are based on clustering techniques, group software entities into different clusters. Although they detect faulty entities from unlabeled data and do not rely on ground truth, labeling the resulting clusters as either defect-prone (or not) is another challenging problem that needs to be addressed.

This paper leverages the *Pareto principle* and uses an **outlier-based defect prediction technique** to identify defective software components. Outlier detection is a Machine Learning technique that identifies objects with very different characteristics compared to the common characteristics observed for all objects [31]. Outlier detection methods can be categorized into two groups, *i.e.*, proximity-based methods and clustering-based methods. We use a proximity-based approach that detects outliers based on (di)similarity of objects. The *Pareto principle* states that 80% of the consequences originate from 20% of the causes. In the context of software defects, it means that 20% of the software modules contain 80% of the faults [32]–[37]. We *hypothesize* that 20% of the software modules may exhibit different characteristics than others and pinpoint the defect-prone components. We use five well-known proximity-based outlier detection techniques, *i.e.*, *K-Nearest Neighbor* (KNN), *Local Outlier Factor* (LOF), *Local Distance-Based Outlier Factor* (LDOF), *Local Outlier Probabilities* (LOOP) and *Angle-Based Outlier Detection* (ABOD) in order to detect the outliers, *i.e.*, the so-called "vital modules" [34]. We evaluate our approach on 16 projects from the PROMISE and GitHub defect prediction data repositories. Selected projects are from various domains with different defect distributions and metric sets. First, we evaluate the applicability of Pareto principle on each dataset and then measure the performance of each technique to detect the most

2470-6892/20/$31.00 ©2020 IEEE

defect-prone components, *i.e.*, the small number of more faulty ones. The evaluation results indicate that the Pareto principle is applicable for all of the projects, and the KNN outlier detection method is able to detect up to 94% of expected defects in 20% of modules.

We compare our approach with the state-of-the-art unsupervised and supervised defect prediction models. In particular, we first compare our results with the recent unsupervised works by Zhang et al. [38] (*"Cross-project Defect Prediction Using a Connectivity-based Unsupervised Classifier"* published at ICSE'16) as well as the work by Yan et al. [39] (*"File-Level Defect Prediction: Unsupervised vs. Supervised Models"* published at ESEM'17). We also compare our approach with the supervised approach proposed by Agrawal and Menzies [40] (*Is "Better Data" Better Than "Better Data Miners"?*, published at ICSE'18). We find out that our outlier-based defect prediction method performs better than other unsupervised methods. It has a mean F-measure value of 37%, compared to the 11% and 31% in other unsupervised approaches. Furthermore, it achieves comparable results with other supervised methods. The evaluation results indicate that our approach is a good alternative for highly tuned supervised methods that rely on labeled training data.

The key novelty of this work is the use of an outlier detection technique to build a defect prediction model. To the best of our knowledge, no prior works have used outlier detection models for software defect prediction. After the review process, we will include the link to our GitHub repository that releases all of our datasets and the scripts to facilitate the reproducibility of research findings. The main contributions of this paper are:

- An empirically grounded work that demonstrates how the Pareto principle could be leveraged for defect prediction.
- An investigation of using five outlier detection techniques (with different configuration) for software defect-prediction.
- A comparison of an outlier-based defect prediction approach with the state-of-the art supervised and unsupervised defect prediction techniques.

The remaining sections of this paper are organized as follows: Section II provides related works and Section III lists the research questions and the methodology used to address them. Section IV conveys the experiment results and Section V elaborates on the results to highlight the key take-aways. Section VI reviews the threats to the validity and Section VII presents the concluding remarks.

II. Related Work

This section discusses previous works on software defect prediction which are based on supervised and unsupervised learning approaches:

A. Supervised Defect Prediction Methods

1) Within-Project Defect Prediction: Most initial studies on software defect prediction used multiple releases of a project. They built prediction models on a project and evaluated the model on the same project. Basili et al. [1] assessed the usefulness of CK metrics [2] for predicting fault-prone classes in a management information system. Using logistic regression, these object-oriented (OO) metrics could provide high performance in fault prediction. Briand et al. [3] also evaluated the effectiveness of CK metrics and other OO design metrics to predict faulty components in an industrial project. To evaluate the prediction model they performed 10-cross validation. Gyimothy et al. [4] evaluated the applicability of the CK metrics to predict the number of bugs in classes. Their experiment on seven versions of Mozilla showed that CBO was the best metric in predicting the fault-proneness of classes. Nagappan et al. used code churn, LOC, and code complexity metrics from Windows XP to estimate the post-release failure-proneness of Windows Server 2003 [5]. Ostrand et al. [6] used code metrics such as LOC and file change history to predict the number of faults in a multiple release software system. They used a negative binomial regression model for fault prediction. Denaro et al. [7] used regression models and data from the Apache 1.3 project to predict defects on the Apache 2.0 project. These studies were conducted in the context of project defect prediction. However, these approaches are unsuitable for projects that do not have historical data available to be used to train the prediction models. Therefore, some researchers proposed cross-project defect prediction models.

2) Cross-Project Defect Prediction: In cross-project defect prediction studies, prediction models were created based on one or more projects and the models were evaluated on other projects. Zimmerman et al. [8] found that among 622 cross-project experiments only 3.4% worked. Turhan et al. [9] showed that cross-company defect predictors could not outperform within-company defect predictors. Rahman et al. [10] evaluated cross-project predictors based on cost-sensitive measures rather than usual classification measures. They showed that inspection of a smaller fraction of the code – in cross-project defect prediction- is as good as within-project defect prediction and better than a random model. Canfora et al. [11] proposed multi-objective cross-project predictor based on the Rahman et al.'s study. They found that multi-objective predictors perform better than single-objective in cross-project defect prediction. He et al. [12] and Herbold [13] used distributional characteristics of datasets to select suitable training data for projects without historical data. Singh and Verma [14] showed that cross-project predictors that were built using design metrics are good predictors for software faulty modules. Panichella et al. [15] proposed a combined approach based on different classifiers that improve the performance of cross-project predictors. Xia et al. [41], Ryu et al. [16] and Li et al. [29] proposed methods to improve the performance of cross-project defect predictors. Kamei et al. [17] proposed cross-project predictors that identify source code changes that have a high risk of introducing a defect. The review reveals that cross-project defect prediction studies have challenges of selecting suitable training data because of heterogeneity [42] and used different methods to improve the performance of these kinds of predictors.

B. Unsupervised Defect Prediction Approaches

Unsupervised approaches try to detect defective components from unlabeled data. They used clustering algorithms to capture software defect clusters. Zhong et al. [18] used k-means and Neural-Gas clustering algorithms to cluster software modules into a small number of coherent group. Then, the software engineering expert inspected different clusters to label them as either fault-prone or not fault-prone. Their results showed that this unsupervised method achieves comparable classification accuracies with other classifiers. Bishnu and Bhattacherjee [19] used k-means algorithm for fault prediction when the fault data for modules are not available. They used Quad Tree-based method and the concept of clustering gain to assign the appropriate initial cluster centers. They showed that the performance of the Quad-tree based fault predictor is comparable with the supervised learning approaches. Park and Hong [20] built unsupervised models for fault prediction using clustering algorithm. They tried to solve the issue of clustering algorithms that is the number of clusters, by using Expectation–Maximization (EM) and Xmeans, which determine the number of clusters automatically. Zhang et al. [30] proposed connectivity-based unsupervised classifier using spectral clustering. They considered the connectivity among software entities based on similarity between metric values. They showed that this unsupervised approach achieves impressive performance in a cross-project defect prediction. In these studies, labeling the clusters as defect-prone or non-defect-prone is a challenging problem. Zhang et al. [38] tried to solve the labeling issue by using average metrics value in each cluster. They considered the cluster with higher average value as defective. Fu and Menzies [43] and Yang et al. [28] used unsupervised approaches, which are not based on clustering algorithms, for change-level defect prediction. Yan et al. [39] used the same approach for file-level defect prediction. Despite the importance and ease of use of unsupervised approaches, limited studies have been conducted in this area.

III. Experiment Design

The approach proposed by this paper relies on the idea that the *Pareto principle* can be applicable in the context of software defect prediction, where a small number of modules contain most of the observed defects. Leveraging the *Pareto principle*, we further hypothesize that these modules may have different characteristics when compared to other modules in a software project. Therefore, we develop an approach using outlier detection techniques to identify them and thereby predict the software defects.

A. Data Collection

We examine data from two publicly-available defect prediction datasets: PROMISE [44] and GitHub [45]. We chose to evaluate our approach using datasets that are publicly-available to reduce the replicability concerns [46]. A brief description of each dataset and selected metrics:

- The PROMISE dataset [44] contains open source Java projects and has 20 object-oriented metrics [2], [47]–[51] at file level. It has been used widely in defect prediction studies. We selected 10 projects with different defect distribution to evaluate our outlier-based approach. We selected the project releases in PROMISE that have been used in recent defect prediction studies [38]–[40].
- The Github dataset [45] is created by gathering information from open source Java projects on GitHub. This dataset covers projects from different domains and has different sets of metrics. We used size, documentation, object-oriented, complexity and code duplication metrics at class level from this dataset [52]. We selected six projects some of which used in prior works [53] from different domains including Android library, language processing, search engine, Java framework, database, and data processing platform with varying sizes ranging from 6 KLOC to 420 KLOC. For each project we selected a version that had larger number of defects.

All projects are from different domains, have diverse size and varied percentage of defective components. Furthermore, the two datasets have different sets of metrics which are calculated at different granularity levels. Therefore, we evaluate our approach and compare it with other approaches by using different sets of metrics at file and class granularity levels. Details of the selected projects are provided in Table I that shows the total number of modules (*#Modules*) in each project, the lines of code (*#LOC*), the number of defective modules (*#Defective Modules*) and the ratio of them (*%Defective Modules*).

B. Research Questions

While developing an outlier-based software defect prediction approach, we investigate five research questions:

RQ1 ***Does the software defect distribution in our datasets follow the Pareto principle?***

The Pareto principle has been evaluated on a limited number of projects in a few previous studies in terms of defect distribution [32]–[37]. They have demonstrated that the fundamentals of the Pareto principle in software defects distribution hold, but the actual percentage of "vital" and "trivial" modules can vary [34]. Since this work began with the idea that we can rely on the Pareto principle for defect prediction, this research question investigates whether the Pareto distribution is applicable to our datasets or not.

RQ2 ***Can an outlier detection technique find buggy modules that account for 80% of the defects in a project?***

In practice, software engineers have to make a trade-off between the accuracy of a model and its cost. This research question investigates to which extent an outlier-based defect prediction technique can be successful. Granted that these techniques achieve suitable performance (in terms of precision and recall) that relieves the need for manually curating a training dataset for defect prediction, which is labor-intensive and costly.

RQ3 ***Which outlier detection technique is more accurate for defect prediction?***

TABLE I: Projects used for evaluation

Dataset	Project	#Modules	LOC	#Defective Modules	%Defective Modules
PROMISE	Ant 1.7	745	208653	166	22.3
	Arcilook 1.0	234	31342	27	11.5
	Camel 1.0	339	33721	13	3.8
	Ivy 2.0	352	87769	40	11.4
	Jedit 4.0	306	144803	75	24.5
	Log4j 1.0	135	21549	34	25.2
	Poi 2.0	314	93171	37	11.8
	Tomcat 6.0	858	300674	77	9
	Xalan 2.4	723	225088	110	15.2
	Xerces 1.3	453	16795	69	15.2
GitHub	Android 2013-07	98	6294	17	17.3
	Antlr 2014-02	479	49090	21	4.4
	ElasticSearch 2014-02	5908	420659	678	11.5
	Junit 2010-04	658	14988	22	3.3
	OrientDB 2013-12	1847	221572	280	15.2
	HazelCast 2014-05	3413	193296	380	11.1

Prior studies suggest that easy-to-use classifiers, such as Naive Bayes and Logistic Regression [27], [54], tend to perform well in defect prediction. In this question, we investigate whether this conclusion hold for an unsupervised outlier-based defect prediction approach that uses different techniques such as distance-based, density-based and angle-based, and which technique performs better compared to others.

RQ4 ***How does a defect predictor built based on an unsupervised outlier detection technique perform in comparison to the state-of-the-art unsupervised approaches?***

Limited number of studies used unsupervised approaches for software defect prediction to address the challenge of collecting training data. In this part of the research, we investigate the performance of the proposed outlier-based software defect predictor in comparison to the predictors built with other unsupervised approaches [38], [39].

RQ5 ***Can a defect predictor built based on an unsupervised outlier detection technique perform well enough to provide comparable results with the state-of-the-art supervised approaches?***

Supervised approaches have been widely used in software defect prediction. Unsupervised approaches usually underperform supervised approaches. This research question investigates whether the performance of the proposed outlier-based software defect predictor is comparable to the predictors built with supervised approaches [40].

We performed a series of experiments to answer these five research questions. The specific methods used to answer each research question are further explained in the next subsections.

C. RQ1: Evaluating the Pareto Principle in our Dataset

To investigate whether the Pareto principle applies to our dataset, we used Alberg diagrams [55], [56]. The concept of the Alberg diagram is that, for each project in our dataset, if we sort software modules in decreasing order with respect to the percentage number of defects (i.e., the total *number of defects in a file* f_i divided by the total *number of defects within the project*), then, we can plot the cumulative percentages of defects for different percentages of modules.

D. RQ2: Outlier-Based Defect Detection

This paper uses proximity-based outlier detection approaches to detect vital few modules that contain most of the defects. Proximity-based approaches consider an object as outlier if the proximity of the object to its nearest neighbors significantly deviates from the proximity of other objects to their nearest neighbors. Since proximity-based approaches do not require any training data, they are considered as unsupervised methods [31]. There are three types of proximity-based approaches which are *distance-based*, *density-based* and *angle-based* techniques. To investigate the feasibility of using outlier-based detection approaches for creating defect prediction models, we used five outlier detection techniques:

- **K-Nearest Neighbor (KNN)**: It is a distance-based technique that measures the degree of outlier-ness of a data object based on the distance d to its *k-th* nearest neighbor [57]. Each data point is ranked based on its distance to its *k-th* nearest neighbor. The top N points in this ranking are considered as outliers.
- **Local Outlier Factor (LOF)**: It is a density-based outlier detection technique [58]. It captures relative degree of isolation for each data object and is based on the density around the data object and its k-nearest neighbors.
- **Local Distance-based Outlier Factor (LDOF)**: It considers the relative location of an object to its K nearest neighbors to determine the degree to which the object deviates from its neighborhood [59]. For each data object, LDOF is calculated as KNN distance divided by KNN inner distance where KNN distance is the average distance of the data object to its K nearest neighbors and KNN inner distance is average distance among its K nearest neighbors.
- **Local Outlier Probabilities (LoOP)**: It is a density-based outlier detection technique that calculates the probability of outlierness for each data object [60].
- **Angle-Based Outlier Detection (ABOD)**: It is an angle-based outlier detection technique that considers the variance in the angles between the difference vectors of a point to its nearest neighbors [61]. We use the improved

version of ABOD algorithm which is called FastABOD and is suitable for large datasets.

For the distance-based approaches, we used the *Euclidean distance* to measure the separation between neighbors. Furthermore, the K parameter for each of these algorithms was experimentally set to 10, 30, 50, 100 and 200. The performance of the algorithms did not improve for larger parameter values like 100 and 200. Therefore, we excluded 100 and 200, and used **10**, **30**, **50** as parameter values.

Our approach encompasses two steps: (1) We use code metrics and rely on various proximity-based approaches (e.g. Euclidean distance) to calculate the dissimilarity between each module and other modules in its neighborhood. (2) We rely on the generated score by these algorithms to identify the non-conformist modules. We sort these software modules in descending order based on their outlier scores and consider the top 20% of these modules as defective modules. We performed the experiments on 16 project releases that are described in Table I. We conducted a total of **240** experiments (16 datasets $\times$ 5 outlier detection techniques $\times$ 3 parameters) to evaluate the performance of the outlier-based detection for software defect prediction.

Leveraging the Pareto principle, the top 20% of the ranked modules were considered as more defective-prone elements in each experiment and it is expected that 80% of the defects will be contained in these top-ranked modules. Therefore, we propose the Recall$_{of\ 80\%}$ as performance measure to evaluate if outlier detection techniques can predict the majority of defects. We use Recall$_{of\ 80\%}$ because our goal is *to examine if a simple approach based on outlier detection can help programmers to find the majority of bugs (those 20% "vital modules")*. However, only this research question will rely on Recall$_{of\ 80\%}$ and the remaining research questions in the paper will rely on recall.

The Recall$_{of\ 80\%}$ is defined as

$$\text{Recall}_{of\ 80\%} = \frac{\text{NODD}}{0.80\ \times\ \text{TND}} \tag{1}$$

where NODD represents the number of detected defects and TND shows the total number of defects.

E. RQ3: Most Accurate Outlier Detector

To answer this research question, we compare the performance of different predictors using five different outlier detection mechanisms. To select the best performing outlier detection technique, we used the Recall$_{of\ 80\%}$ in Equation 1, and the Precision, Recall and F-measure in Equations 2, 3, and 4 as performance measures. In Equation 2, NODDM represents the number of detected defective modules and NOMPD shows the number of modules predicted as defective. In Equation 2 TNDM represents the total number of defective modules.

$$Precision = \frac{\text{NODDM}}{\text{NOMPD}} \tag{2}$$

$$Recall = \frac{\text{NODDM}}{\text{TNDM}} \tag{3}$$

$$F - measure = 2 * \frac{\text{Precision*Recall}}{\text{Precision+Recall}} \tag{4}$$

F. RQ4: Comparing with State-of-the-Art Unsupervised Approaches

We compare our approach with two state-of-the-art unsupervised approaches. The one that uses connectivity-based clustering technique for defect prediction and the one that uses the simple sorting approach:

1) *"Cross-project Defect Prediction Using a Connectivity-based Unsupervised Classifier"* by Zhang et al. [38] at ICSE'16.
2) *"File-Level Defect Prediction: Unsupervised vs. Supervised Models"* by Yan et al. [39] published at ESEM'17.

1) Selection Rationale: We chose the work by Zhang et al. [38] because they propose a simple approach for labeling clusters as defective or non-defective which is a challenging problem for clustering-based defect prediction approaches. Furthermore, their clustering-based approach has good performance for both cross-project and within-project defect prediction. We also selected the Yan et al. [39] that was one of the first works that emphasized the value and ease of use of simple unsupervised techniques in file-level bug predictions. The approach used in this study is valuable, because it demonstrates that defect prediction is possible with simple methods [43]. The underlying approaches in each of these papers are briefly summarized in the next subsections.

2) Overview of Zhang et al. Work: This article uses spectral clustering that is a connectivity-based clustering approach for defect prediction. Their unsupervised approach is based on the findings that defective entities tend to cluster around the same neighbourhood [62]. Therefore, they partition software modules into two clusters based on the similarity between them. They create a software graph by considering software modules as nodes and the similarity between them as edge weights. Then, they perform spectral clustering on the software graph and use the relationship between metrics and defect proneness to label clusters. Gaffney [63] show that larger files are more defect-prone compared to smaller files. Therefore, they calculate the summation of metrics for each software module and consider the cluster with a higher average summation as defective.

3) Overview of Yan et al. Work: This article uses the unsupervised model proposed by [39]. Their approach is based on Koru et al.'s finding that smaller modules are proportionally more defect-prone and hence should be inspected first [64]. Their unsupervised model is built by ranking files in descending order according to the reciprocal of their corresponding raw metric values. They built different unsupervised models based on the value of each metric. Specifically, for each code metric M, the corresponding model is $R(i) = 1/M(i)$ where R is the predicted risk value for file i. The unsupervised models which were based on Average Method per Class (AMC) and Response for Class (RFC) metrics had the best recall of defective files. We compare our best model KNN-50 with their best unsupervised model that is based on the

RFC metric (which is available in all datasets). The Precision, Recall and F-measure are used as performance measures.

G. RQ5: Comparing with State-of-the-Art Supervised Approaches

We compare our approach with the state-of-the-art supervised approach entitled *Is "Better Data" Better Than "Better Data Miners"*, published at at ICSE'18 and authored by Agrawal and Menzies [40].

1) Selection Rationale: We chose the work by Agrawal and Menzies [40] because it uses a complex auto tuning approach to enhance a number of supervised bug prediction models. Their approach relies on fixing the issues in imbalanced datasets and therefore creates better training data for bug prediction algorithms.

2) Overview of Agrawal and Menzies Work: This paper [40] tunes parameters for SMOTE [65] which is a widely used approach for handling the data imbalance in software engineering. SMOTE handles class imbalance by sub-sampling the majority class and over-sampling the minority class. The paper tunes the SMOTE parameters for each dataset to improve the performance of defect predictors. The underling assumption of this work is that there are enough data from a given project to optimize the bug prediction models for that project. The approach called SMOTUNED uses differential evolution for finding the best set of parameters. SMOTUNED uses performance measures Precision, Recall and F-Measure as fitness function and tries to maximize them on the same datasets. F-measure is used as performance metric while comparing our results with this approach.

IV. Results

The results obtained for each research question listed in Section III-B are provided in the next subsections:

A. RQ1: Pareto Principle on the selected datasets

To test the applicability of the Pareto principle, the Alberg diagram of all projects are created. Table II shows information obtained from Alberg diagrams which is percentage of defects that exist in 20% of modules for each project. All projects follow the Pareto principle where 20% of their modules cover over 80% of the software defects. In particular, for **13** of these projects, the 20% of the modules covered ***all*** the defects in the project. In the *Ant*, and *Jedit* projects, 20% of modules covered between 90% to 99% of the defects. In Log4j, 20% of modules covered about 80% of the defects. All of the projects confirmed the applicability of the Pareto principle where the majority of the defects existed in close to 20% of the modules. Therefore, we could leverage the Pareto principle for the aforementioned projects to detect software components that contain the majority of defects.

RQ#1 Findings:

TABLE II: The percentage of defects in 20% of software modules.

% Defects in 20% of Modules	Projects
100	Arcilook, Camel, Ivy, Poi, Tomcat, Xalan, Xerces, Android, Antlr, ElasticSearch, Junit, OrientDB, HazelCast
90-99	Ant, Jedit
80-89	Log4j

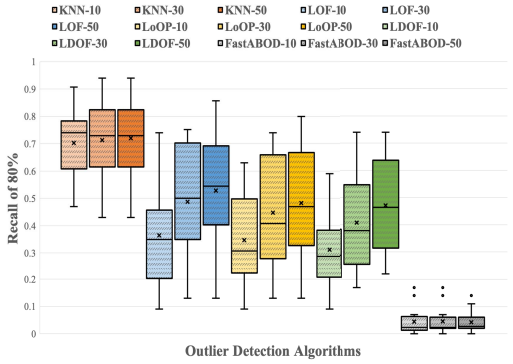


Fig. 1: Boxplot for the $Recall_{of\ 80\%}$ for each outlier detection approach with K values 10, 30, and 50.

— The Pareto principle is applicable for the software defect distribution in the PROMISE and Github projects tested.

B. RQ2: Outlier-Based Defect Detection

We hypothesized that 20% of modules (the "vital modules") shown in RQ1 would exhibit different characteristics compared to other modules. Therefore, we use outlier detection techniques that could detect the small portion of modules that contain the majority of faults.

To answer RQ2 we calculate the $Recall_{of\ 80\%}$. Because we consider 20% of the modules in each project as faulty, we expect that about 80% of the faults can be detected in these vital modules. Therefore, we use $Recall_{of\ 80\%}$ as performance measure to detect how successful the proposed oulier-based prediction models are in detecting these vital modules. Fig. 1 shows the boxplot for the $Recall_{of\ 80\%}$ for all projects. The overall observation for the outlier detection techniques is that an increase in "K" parameter leads to an increase in $Recall_{of\ 80\%}$. In particular, our highest $Recall_{of\ 80\%}$ is 94% when using KNN with a *K* value of *30* or *50*. We evaluated greater values of the *K* parameter (100, 200), but they did not improve the performance of the models. Therefore, we report the results obtained by setting the value of the *K* parameter up to 50. The KNN with a K value of 50 exhibits a high Recall$_{of\ 80\%}$ on the projects *Ant*, *Ivy*, *JEdit*, *Tomcat*, *Android*, and *Antlr*. Their Recall$_{of\ 80\%}$ are **0.84**, **0.83**, **0.94**, **0.82**, **0.80**, and **0.88**, respectively. As shown in box plot of the best model (KNN-50) the Recall$_{of\ 80\%}$ is above 0.83 for 25% of the

projects, above 0.72 for 50% of the projects and above 0.63 for 75% of the projects.

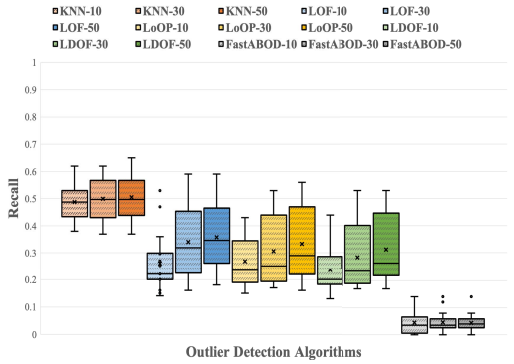
RQ#2 Findings: The defect predictor built with an outlier detection technique is simple and does not require any training data or optimizations and can detect the majority of the bugs rooted in *nonconformist* (outlier) modules. The best performing KNN model (with $k = 50$) detects 94% of the defects. The Recall$_{of\ 80\%}$ for KNN-50 is above 72% for half of the projects and above 83% for a quartile of the projects.

C. RQ3: Best Performing Outlier Detection Technique

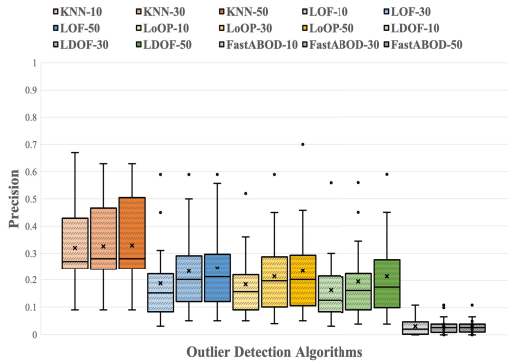
The proposed defect prediction model which is based on outlier detection techniques can be used as a binary classification model. It classifies 20% of the modules with highest outlier scores as faulty and the remaining modules as non-faulty. To make a comprehensive evaluation for selecting the best performing outlier detection technique, in addition to Recall$_{of\ 80\%}$, we consider Precision, Recall and F-measure as performance measures. Fig. 2 shows the box plots of these performance measures for different outlier detection techniques. The overall trend is similar to the trend observed in Fig. 1. An increase in "K" parameter leads to an increase in the Recall, Precision and F-measure in all outlier detection techniques. KNN, which is a simple distance-based outlier detection algorithm, has higher Recall, Precision and F-measure in detecting faulty components. FASTABOD which is an angle based technique perform poorly in defect prediction. Density-based outlier detection techniques such as LOF and LoOP perform better than angle based approach, but weaker than KNN. KNN with $K = 50$ performs better than other techniques while identifying defective modules. The maximum value of Recall, Precision and F-measure for KNN-50 is 0.65, 0.63 and 0.60 and the mean values are 0.51, 0.33 and 0.37. We used Wilcoxon Signed-Rank Test to see if there is a statistically significant difference between the performance of the KNN-50 technique and the other outlier detection techniques. For the comparison, we considered all the models with $K = 50$. The p-values of comparing the four performance measures (Recall$_{of\ 80\%}$, Precision, Recall and F-measure) between KNN and other outlier detection techniques are shown in Table III. The results show that all p-values are <0.05 that shows KNN-50 performs significantly better than the other outlier detection techniques in terms of Recall$_{of\ 80\%}$, Precision, Recall and F-measure.

Based on the results of the first two questions we conclude that:

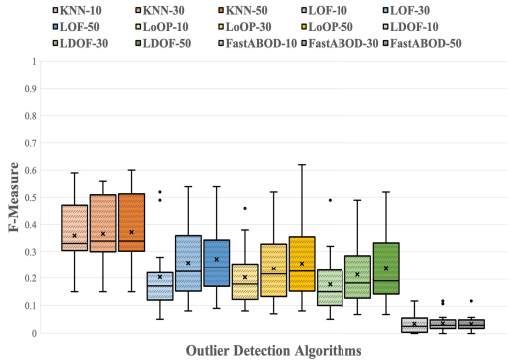
RQ#3 Findings: The KNN outlier detection technique achieves promising results for software defect prediction and its performance is significantly better than other more complex outlier detection techniques.



(a) Recall



(b) Precision

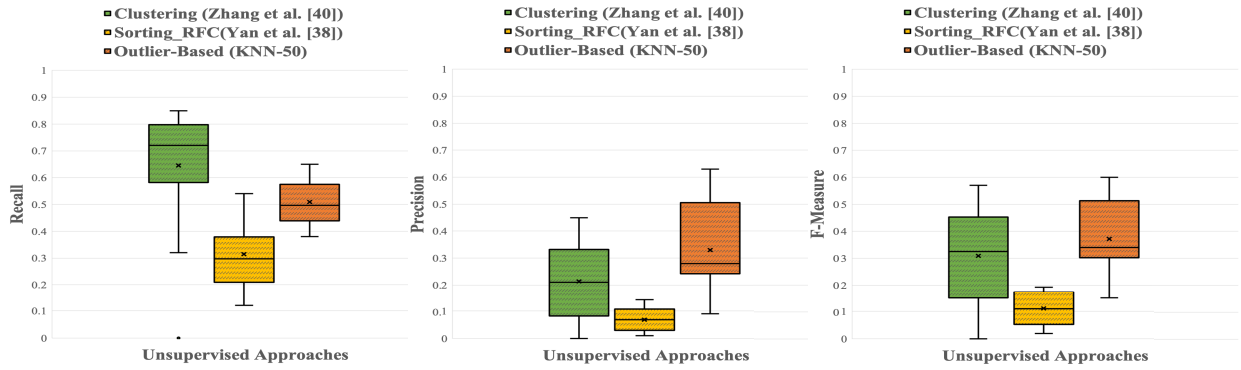


(c) F-Measure

Fig. 2: Boxplots of the Precision, Recall and F-measure for each outlier detection with K values 10, 30, and 50.

TABLE III: Wilcoxon signed-rank test for KNN vs. other outlier detection techniques ($K = 50$ for all the models).

		LOF	LoOP	LDOF	FAST ABOD
KNN	Recall of 80%	0.00096	0.00174	0.00064	0.00044
	Precision	0.00096	0.00148	0.00064	0.00044
	Recall	0.00096	0.00094	0.00064	0.00044
	F-measure	0.00096	0.00124	0.00064	0.00044



(a) Recall (b) Precision (c) F-Measure

Fig. 3: Comparison results of our best outlier-based approach (KNN-50) with Zhang et al. [38] and Yan et al. [39] unsupervised approaches

D. RQ4: Comparison with the State-of-the-art Unsupervised Defect Prediction Approaches

In this section, we compare our outlier-based defect prediction approach with two state-of-the-art unsupervised approaches. An unsupervised work presented in *"Cross-project Defect Prediction Using a Connectivity-based Unsupervised Classifier"* by Zhang et al. [38] published at ICSE'16 and the other work presented in *"File-Level Defect Prediction: Unsupervised vs. Supervised Models"* by Yan et al. [39] published at ESEM'17. The comparison results are shown in Fig. 3. The results show that with regard to Recall, Precision and F-measure Yan et al.'s [39] simple sorting-based approach performs weakly in comparison with our approach and Zhang et al. [38] approach. Recall, *Precision* and *F-measure* of their unsupervised approach are very low with mean values of 0.31, 0.07 and 0.11, respectively. The experiment results reject the claim that decades of study on defect prediction were complicated needlessly [28]. As shown in Fig. 3, Machine Learning-based approaches perform significantly better than the simple sorting-based approach.

Zhang et al. [38] clustering-based approach has better *Recall* in comparison to our best outlier-based approach. However, low values of *Precision* show that their approach produce a lot of False Positives (FP) that could increase the effort required for code inspection. F-measure values show that our simple outlier-based approach (KNN-50) performs better than Zhang et al. [38] clustering-based approach. While the mean value of *F-measure* among all datasets for our outlier-based model is 0.37, Zhang et al. [38] approach has a mean value of 0.31. *F-measure* value for our approach was 0.60 in best case which was 0.57 and 0.19 in Zhang et al. and Yan et al. approaches. The p values of the *Wilcoxon Signed-Rank Test* provided in Table IV show that the *F-measure* of the outlier detection approach is significantly better than the state-of-the-art unsupervised approaches [38], [39].

TABLE IV: Wilcoxon signed-rank test for F-measure of outlier-based approach (KNN-50) vs. F-measure of the state-of-the-art unsupervised approaches.

	Unsupervised Approaches	
	Clustering (Zhang et al. [40])	Sorting_RFC (Yan et al. [38])
Outlier-Based (KNN-50)	0.03572	0.00064

RQ#4 Results (Comparison with state-of-the-art unsupervised approaches):

— The proposed simple outlier-based approach (KNN-50) performs significantly better than state-of-the-art unsupervised approaches. While the mean value of *F-measure* for Zhang et al. [38] and Yan et al. [39] studies are 0.31 and 0.07 respectively, the mean value of *F-measure* for our approach (KNN-50) is 0.37.

— Weak performance of Yan et al. [39] sorting-based approach on the various selected datasets (which contains different projects from various domains with different metric sets) rejects the claim that a simple sorting based approach can perform as good as or better than more complicated approaches [28].

E. RQ5: Comparison with a State-of-the-art Supervised Approach

We compare our best outlier-based prediction model (the KNN-50) with the labour-expensive, supervised and tuned models presented in *Is "Better Data" Better Than "Better Data Miners"?* by Agrawal and Menzies [40] at ICSE'18. To perform the comparison, we compared the KNN-50 with 6 supervised models that have been used by Agrawal and Menzies [40]. We used Random Forest (RF), Logistic Regression (LR), KNN Classifier, Naive Bayes (NB), Decision Tree (DT), and Support Vector Machines (SVM) as supervised models. These classifiers were ranked from *best* to *worst* classifiers for

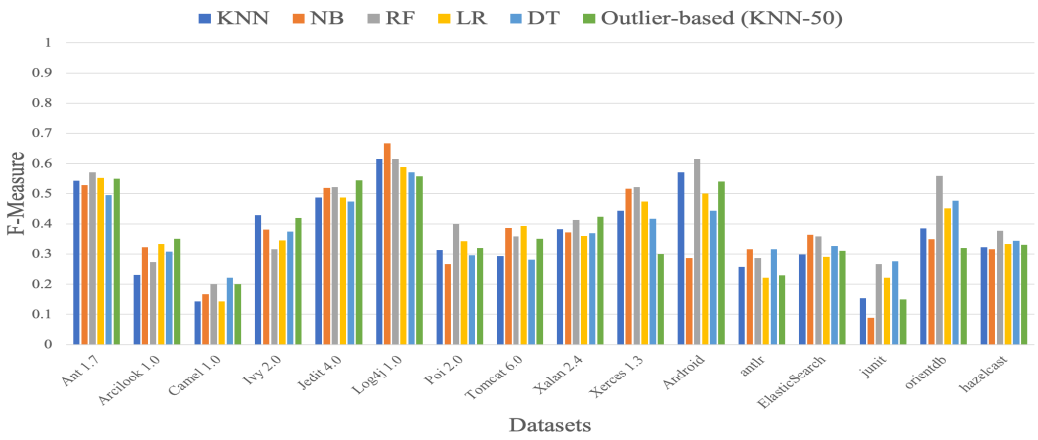


Fig. 4: Comparing outlier-based approach (KNN-50) with the Agrawal and Menzies approach [40]

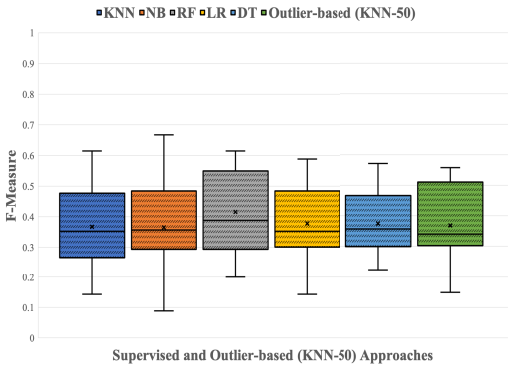


Fig. 5: Boxplots for comparing outlier-based approach (KNN-50) with Agrawal and Menzies supervised approach [40]

defect prediction [66]. KNN classifier finds K-most similar instances from training data and selects the class label with high frequency among K instances for prediction. However, our KNN outlier-based approach considers the euclidean distance between the Kth nearest neighbor as outlier score. During the experiments, we found that running SVM for larger datasets with auto tuning required a lot of time, therefore we excluded it from our experiments. Agrawal and Menzies [40] tuning approach shuffles the dataset and then selects the train and test data for cross-validation. We considered the median of the F-measure value obtained from 15 shuffling as their final measure for each supervised model as mentioned in their study. The comparison results are shown in Fig. 4. The performance of the KNN outlier detection approach varies on different datasets. The boxplots of the supervised approaches and our best model (KNN-50) are shown in Fig. 5. The mean value of the *F-measure* for our outlier-based approach is 0.37 which is in order 0.37, 0.36, 0.41, 0.38 and 0.37 for KNN, NB, RF, LR and DT methods. To see if there is any significant difference between our approach and their highly tuned supervised approaches, we compared the F-measure values using the *Wilcoxon Signed-Rank Test*. The p-values

TABLE V: Wilcoxon signed-rank test for Agrawal and Menzies [40] vs. our outlier-based approach (KNN-50)

	Supervised Approaches				
	KNN	NB	RF	LR	DT
(KNN-50)	0.79486	0.87288	0.06876	0.96012	0.96012

that are provided in Table V show that there is not any significant difference between our results and their highly improved supervised results (all p-values>0.05). The simple outlier detection approach, which does not require any training data, can perform as good as the supervised defect prediction approaches that are highly tuned to improve the F-measure in within-project experiments.

RQ#5 Results (comparison with a supervised state-of-the-art approach):

— There is no statistically significant difference between the performance of Agrawal and Menzies [40] and our proposed outlier-based defect prediction technique. However, by using our approach, there is a significant reduction in the effort required to collect and label training data sets and tune prediction algorithms.

V. Discussion

Decades of research have been devoted to develop bug detection techniques using complex supervised learning methods. However, the cost of collecting training data is a great barrier to start using defect prediction methods in the industry [9], [67]. In practice, software engineers have to make a trade-off between the model accuracy and the cost of building a successful defect predictor. They need to deal with various challenges, including obtaining accurate training data to fine-tune a complex learning algorithm. In this paper, we want to check whether a predictor based on an outlier detection

technique that does not require training data, is able to achieve an acceptable defect prediction performance or not?

While there was no statistically significant difference between our work and Agrawal and Menzies's work [40], our approach is significantly simpler and requires no training data or tuning. Therefore, during the early stages of software development where there are not enough bug reports or in cases that developers do not have labeled data, the best approach could be to focus on source files that are nonconformist and appear as outliers. These files could count for a significant number of the defects in the system.

Agrawal and Menzies work [40] used SMOTUNED auto tuning to create better training data and therefore improve the recall of the state-of-the-art works by 20%. Similar results were achieved using the outlier detection method, without a tuning need. We do not argue against supervised learning techniques for bug prediction, however in situations where the ground truth data is limited, outlier-based bug prediction method can be a viable approach and bug prediction process can be simplified.

Zhang et al. [38] proposed a clustering based unsupervised learning approach that were among the best models in comparison to other unsupervised and supervised approaches. Yan et al. [39] used a simple file-level bug prediction approach that does not have any complications but, they didn't evaluated the False Positive rate of the approach. We re-implemented and evaluated their approach on other datasets. The experiment results indicate that their approach suffers from low Recall, Precision and F-measure. We show that these approaches could be outperformed by the outlier-based unsupervised defect prediction method. As a future work, we plan to optimize the techniques used in the proposed approach to improve the defect prediction performance further.

VI. Threats to Validity

In this section we discuss threats present in our study:

Internal Validity. The internal validity is related to uncontrolled aspects of the study that may affect the results. The potential threat to the validity of our approach is how well we compared our results with state-of-the-art supervised and unsupervised approaches. To eliminate this threat, first, we implemented our approach on part of the datasets that were used in these studies. For comparison with Agrawal and Menzies study [40] we used the raw data from dump files that were available in their Github repository and executed their application to check if the results are the same or not. For comparing with Yan et al.'s study [39] we re-implemented their approach and calculated recall, precision, and F-measure. Since they only reported recall as a performance measure, by comparing recall we found that our implementation generates the same results.

External Validity. The external validity is related to the possibility to generalize the findings of a study. The most important threat to the external validity of this study is that our unsupervised approach that is based on the Pareto Principle may not be generalized to other projects. To mitigate this, we selected a large number of (16) data sets with different sizes and fault distributions. However, future work might include verifying our unsupervised approach on additional software projects, particularly those which are implemented in different programming languages. Furthermore, we have provided the necessary details and scripts that will make it easy for others to replicate our study.

VII. Conclusions

This work proposes a simple and easy to use unsupervised approach based on an outlier detection technique to predict defect-prone components that contain most of the software defects. Supervised defect prediction models that have been studied in the literature rely on the labeled training datasets that may not be easy to obtain. Unsupervised defect prediction methods relax the requirement of the training data. However, most unsupervised defect prediction approaches that have been proposed so far use clustering algorithms to group the software components into defect-prone and non-defect-prone clusters, and identifying faulty clusters is a key challenge that remains to be addressed.

This paper leverages the Pareto principle to detect 20 percent of components that contain the majority of faults by using outlier detection methods. It uses 16 publicly available projects at the PROMISE and GitHub defect prediction repositories. First, we evaluate the Pareto principle in these projects and show that all projects support the Pareto principle in terms of defect distribution. Then, we evaluate different distance-based, density-based, and angle-based outlier detection techniques to find software components that contain most of the defects. The project modules are ordered in the descending order by their outlier score and the top 20% of the modules are considered as most faulty ones. The KNN outlier detection method (with $K = 50$) performs better than other outlier detection algorithms and could detect up to 94% of the software defects. We compare the performance of our approach with the state-of-the-art unsupervised and supervised approaches. The proposed approach performs significantly better than other unsupervised approaches. Our approach could predict faulty components with a maximum F-measure value of 60% and a mean value of 37% which are 19% and 11% in the sorting-based and 57% and 31% in the clustering-based unsupervised approaches. Furthermore, the experiment results indicate that the performance of our simple outlier detection based approach, which does not require any training data, achieves comparable results with the highly tuned supervised approach proposed by Agrawal and Menzies [40].

References

[1] V. R. Basili, L. C. Briand, and A. W. L. Melo, "Validation of object-oriented design metrics as quality indicators," *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[2] S. R. Chidamber and A. C. F. Kemerer, "metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[3] L. C. Briand *et al.*, *Investigating quality factors in object-oriented designs: an industrial case study. in Proceedings of the 21st international conference on Software engineering*. ACM, 1996.

[4] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[5] N. Nagappan, T. Ball, and B. Murphy, "Using historical in-process and product metrics for early estimation of software failures. in software reliability engineering," in *2006*. ISSRE'06. 17th International Symposium on IEEE, 2006.

[6] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, *Where the bugs are. in ACM SIGSOFT Software Engineering Notes*. ACM, 2004.

[7] G. Denaro and M. Pezzè, *An empirical evaluation of fault-proneness models. in Proceedings of the 24th International Conference on Software Engineering*. ACM, 2002.

[8] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "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, 2009, pp. 91–100.

[9] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[10] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the "imprecision" of cross-project defect prediction," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering ACM: Cary*, N. Carolina, Ed. p, 2012, pp. 1–11.

[11] G. Canfora *et al.*, *Multi-objective Cross-Project Defect Prediction. in 2013 IEEE Sixth International Conference on Software Testing*. Verification and Validation, 2013.

[12] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Software Engineering*, vol. 19, no. 2, pp. 167–199, 2012.

[13] S. Herbold, "Training data selection for cross-project defect prediction," in *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*. ACM, 2013, p. 6.

[14] P. Singh and S. Verma, "Cross project software fault prediction at design phase," *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 9, no. 3, pp. 800–805, 2015.

[15] A. Panichella, R. Oliveto, and A. De Lucia, "Cross-project defect prediction models: L'union fait la force," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 164–173.

[16] D. Ryu, J.-I. Jang, and J. Baik, "A transfer cost-sensitive boosting approach for cross-project defect prediction," *Software Quality Journal*, vol. 25, no. 1, pp. 235–272, 2017.

[17] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, 2016.

[18] S. Zhong, T. M. Khoshgoftaar, and N. Seliya. Unsupervised Learning for Expert-Based Software Quality Estimation. in HASE, 2004.

[19] P. S. Bishnu and V. Bhattacherjee, "Software fault prediction using quad tree-based k-means clustering algorithm," *IEEE Transactions on knowledge and data engineering*, vol. 24, no. 6, pp. 1146–1150, 2012.

[20] M. Park and E. Hong, "Software fault prediction model using clustering algorithms determining the number of clusters automatically," *International Journal of Software Engineering and Its Applications*, vol. 8, 2014.

[21] G. Abaei, A. Selamat, and H. Fujita, "An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction," *Knowledge-Based Systems*, vol. 74, pp. 28–39, 2015.

[22] A. Boucher and M. Badri, *Predicting Fault-Prone Classes in Object-Oriented Software: An Adaptation of an Unsupervised Hybrid SOM Algorithm. in 2017 IEEE International Conference on Software Quality*. Reliability and Security (QRS, 2017.

[23] R. Özakıncı and A. Tarhan, "Early software defect prediction: A systematic map and review," *Journal of Systems and Software*, vol. 144, pp. 216–239, 2018.

[24] W. Zhang, S.-C. Cheung, Z. Chen, Y. Zhou, and B. Luo, "File-level socio-technical congruence and its relationship with bug proneness in oss projects," *Journal of Systems and Software*, vol. 156, pp. 21–40, 2019.

[25] A. Okutan and O. T. Yıldız, "Software defect prediction using bayesian networks," *Empirical Software Engineering.*, vol. 19, no. 1, p. 154–181, Feb. 2014. [Online]. Available: https://doi.org/10.1007/s10664-012-9218-8

[26] A. Okutan and O. Taner Yildiz, "A novel kernel to predict software defectiveness," *Journal of Systems and Software*, vol. 119, pp. 109 – 121, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216300759

[27] R. S. Wahono, "A systematic literature review of software defect prediction: Research trends, datasets, methods and frameworks," *Journal of Software Engineering*, vol. 1, no. 1, pp. 1–16, 2015.

[28] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 157–168.

[29] Z. Li, X. Jing, X. Zhu, H. Zhang, B. Xu, and S. Ying, "On the multiple sources and privacy preservation issues for heterogeneous defect prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 4, pp. 391–411, April 2019.

[30] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 309–320.

[31] J. Han, J. Pei, and M. Kamber, "Data mining: concepts and techniques." Elsevier, 2011.

[32] C. Andersson and A. P. Runeson, "replicated quantitative analysis of fault distributions in complex software systems," *IEEE Transactions on Software Engineering*, vol. 33, p. 5, 2007.

[33] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[34] M. Gittens, K. Yong, and D. Godwin, *The vital few versus the trivial many: examining the Pareto principle for software*, vol. 29, 2005.

[35] T. G. Grbac, P. Runeson, and A. D. Huljenić, "second replicated quantitative analysis of fault distributions in complex software systems," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 462–476, 2013.

[36] M. Hamill and K. Goseva-Popstojanova, "Common trends in software fault and failure data," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 484–496, 2009.

[37] T. J. Ostrand and E. J. Weyuker, *The distribution of faults in a large industrial software system. in ACM SIGSOFT Software Engineering Notes*. ACM, 2002.

[38] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 309–320.

[39] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang, "File-level defect prediction: Unsupervised vs. supervised models," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Nov 2017, pp. 344–353.

[40] A. Agrawal and T. Menzies, "Is better data better than better data miners?: on the benefits of tuning smote for defect prediction," in *Proceedings of the 40th International Conference on Software engineering*. ACM, 2018, pp. 1050–1061.

[41] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "Hydra: Massively compositional model for cross-project defect prediction," *IEEE Transactions on software Engineering*, vol. 42, no. 10, pp. 977–998, 2016.

[42] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 2, pp. 111–147, 2017.

[43] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 72–83.

[44] J. Sayyad Shirabad and T. Menzies, "The PROMISE Repository of Software Engineering Databases." School of Information Technology and Engineering, University of Ottawa, Canada, 2005. [Online]. Available: http://promise.site.uottawa.ca/SERepository

[45] Z. Tóth, P. Gyimesi, and R. Ferenc, "A public bug database of github projects and its application in bug prediction," in *International Conference on Computational Science and Its Applications*. Springer, 2016, pp. 625–638.

[46] G. Robles, "Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 171–180.
[47] R. Martin, "Oo design quality metrics," *An analysis of dependencies*, vol. 12, pp. 151–170, 1994.
[48] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics. in software metrics symposium," vol. 1999, 1999.
[49] J. Bansiya and A. C. G. Davis, "hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.
[50] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995.
[51] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.
[52] R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth, and T. Gyimóthy, "An automatically created novel bug dataset and its validation in bug prediction," *Journal of Systems and Software*, p. 110691, 2020.
[53] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A semantic lstm model for software defect prediction," *IEEE Access*, vol. 7, pp. 83 812–83 824, 2019.
[54] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.
[55] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, 1996.
[56] N. Ohlsson, M. Helander, and C. Wohlin, "Quality improvement by identification of fault-prone modules using software design metrics," in *Proceedings: International Conference on Software Quality*, 1996, pp. 1–13.
[57] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets," *SIGMOD Rec*, vol. 29, no. 2, pp. 427–438, 2000.
[58] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: identifying density-based local outliers," vol. 29, no. 2, pp. 93–104, 2000.
[59] K. Zhang, M. Hutter, and A. H. Jin, "New local distance-based outlier detection approach for scattered real-world data," in *Advances in Knowledge Discovery and Data Mining: 13th Pacific-Asia Conference*. s Springer Berlin Heidelberg: Berlin, Heidelberg. p. 813-822, 2009, pp. 27–30.
[60] H.-P. Kriegel, P. Kröger, E. Schubert, and A. Zimek, "Loop: local outlier probabilities," in *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, 2009, pp. 1649–1652.
[61] H.-P. Kriegel, "M," in *S*. hubert, and A. Zimek, Angle-based outlier detection in high-dimensional data, in Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining ACM: Las Vegas, Nevada, USA. p, 2008, pp. 444–452.
[62] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, "Local vs. global models for effort estimation and defect prediction," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 343–351.
[63] J. E. Gaffney, "Estimating the number of faults in code," *IEEE Transactions on Software Engineering*, no. 4, pp. 459–464, 1984.
[64] G. Koru, H. Liu, D. Zhang, and K. El Emam, "Testing the theory of relative defect proneness for closed-source software," *Empirical Software Engineering*, vol. 15, no. 6, pp. 577–598, Dec 2010. [Online]. Available: https://doi.org/10.1007/s10664-010-9132-x
[65] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.
[66] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 789–800.
[67] A. Tosun, A. Bener, B. Turhan, and T. Menzies, "Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry," *Inf. Softw. Technol.*, vol. 52, no. 11, p. 1242–1257, Nov. 2010. [Online]. Available: https://doi.org/10.1016/j.infsof.2010.06.006