applied sciences

MDPI

Article

Software Defect Prediction Using Heterogeneous Ensemble Classification Based on Segmented Patterns

Hamad Alsawalqah [1], Neveen Hijazi [1], Mohammed Eshtay [2], Hossam Faris [1,*], Ahmed Al Radaideh [3], Ibrahim Aljarah [1] and Yazan Alshamaileh [1]

[1] King Abdullah II School for Information Technology, The University of Jordan, Amman 11942, Jordan; h.sawalqah@ju.edu.jo (H.A.); neveen.hijazi@gmail.com (N.H.); i.aljarah@ju.edu.jo (I.A.); y.shamaileh@ju.edu.jo (Y.A.)

[2] Department of Computer Science, Faculty of Information Technology, Zarqa University, Zarq 13132, Jordan; m.eshtay@fgs.ju.edu.jo

[3] College of Humanities and Sciences, University of Science and Technology of Fujairah, Fujairah 2202, UAE; a.alradaideh@ustf.ac.ae

* Correspondence: hossam.faris@ju.edu.jo

Received: 28 October 2019; Accepted: 25 February 2020; Published: 3 March 2020

Abstract: Software defect prediction is a promising approach aiming to improve software quality and testing efficiency by providing timely identification of defect-prone software modules before the actual testing process begins. These prediction results help software developers to effectively allocate their limited resources to the modules that are more prone to defects. In this paper, a hybrid heterogeneous ensemble approach is proposed for the purpose of software defect prediction. Heterogeneous ensembles consist of set of classifiers of different learning base methods in which each of them has its own strengths and weaknesses. The main idea of the proposed approach is to develop expert and robust heterogeneous classification models. Two versions of the proposed approach are developed and experimented. The first is based on simple classifiers, and the second is based on ensemble ones. For evaluation, 21 publicly available benchmark datasets are selected to conduct the experiments and benchmark the proposed approach. The evaluation results show the superiority of the ensemble version over other well-regarded basic and ensemble classifiers.

Keywords: software defect prediction; ensembles; clustering; segmentation; classification

1. Introduction

Individuals and society increasingly rely on advanced software systems. Because software is intertwined with all aspects of our lives, it is essential to produce reliable and trustworthy systems economically and quickly. In order to ensure the desired software quality at a lower cost, much effort has been invested on software reliability and software quality assurance (SQA) [1,2]. With limited resources, however, this is increasingly being challenged by the rapid growth in size and complexity of today's software. Defective software modules increase the development and maintenance costs and cause customer dissatisfaction [3,4].

Software defect prediction is one of the SQA activities that aims to automatically predict fault-prone software modules using historical software information from an earlier deployment or identical objects, for example source code edit logs [5] and bug reports [6], before the actual testing process begins. Effective defect prediction could help test managers locate bugs and facilitate the allocation of limited SQA resources optimally and economically; thus, it has become an extremely important research topic [7–12]. Commonly, a prediction model is used to predict the defective software modules in one of the three categories: binary class classification of defects [13–16], number of defects/defect density prediction [17–20], and severity of defect prediction [21,22].

Among them, the binary class classification is the most frequently used types of prediction scheme, where software modules having one or more defects are marked as defected and modules having zero defects are marked as non-defected. In this type of defect prediction schema, researchers have explored the use of various classification techniques, including statistical techniques such as Naïve Bayes (NB) [23] and Logistic Regression [24]; supervised techniques such as Decision Tree (DT) [25], Support Vector Machine (SVM) [26], ensemble methods [16,27–29], and Case Based Reasoning [30]; semi supervised techniques such as Expectation Maximization (EM) [31]; and unsupervised techniques such as K-means clustering [32] and Fuzzy clustering [33]. Most of the studies in the literature have used statistical and supervised learning techniques [34].

Although a large number of studies have been conducted to build and evaluate defect prediction models using different classification techniques in the context of binary class classification, still the prediction accuracy of defect prediction techniques is found to be considerably low, with a high misclassification rate [26,34–36]. Looking at these results, one questions the dependability of these techniques for software defect prediction [34,37]. Therefore, it will be important to design more advanced techniques to improve the performance of defect prediction models [34,38].

In this work, a hybrid heterogeneous ensemble approach is proposed for improving the accuracy of software defect prediction. The core argument for this approach is to develop expert and robust classification models of different natures based on groups of similar points. In other words, the classification models are of different machine learning types like lazy classifiers, decision trees, naïve bayes, and ensembles. While on the other hand, similar points refer to a group of points that are as close as possible according to a similarity measure like the euclidean measure. These groups of data are generated using a clustering stage. Unlike most of the previous works that generate general models for all data, this work aims to develop several expert models based on the characteristics of the data. Two versions of the proposed approach are developed and experimented. The first is based on simple classifiers (i.e., k-Nearest Neighbour (*k*-NN), NB, and DT), and the second is based on ensemble ones (i.e., Bagging, Adaptive Boosting (AdaBoost), Random Forest (RF), and XGBoost (XGB)). Extensive experiments based on 21 well-known benchmark datasets are conducted to evaluate the proposed approach.

The remainder of this article is organized as follows: The next section presents related work on defect prediction. The preliminaries of the algorithms utilized in the proposed approach are given in Section 3. Section 4 presents the proposed hybrid heterogeneous ensemble approach for software defect prediction. Section 5 discusses the model evaluation metrics, and Section 6 presents the benchmark datasets specifications. Section 7 is devoted to the benchmarking experiments and discusses their respective results. Finally, Section 8 draws conclusions and describes promising directions for future work.

2. Related Work

During the last two decades, software defect prediction problem became a noteworthy research topic, increasingly catching the interest of researchers. A software defect prediction model can be used to classify software modules into defected or non-defected (binary class classification), to predict the number of defects in a software module, or to predict the severity of the defects. In the context of binary class classification, hundreds of different defect prediction models have been published. To build these models, researchers have used various classification techniques to build the defect prediction models such as Logistic Regression [24], NB [23], SVM [26], ANN [39], Genetic Programming [40], Ant Colony Optimization [14], Particle Swarm Optimization [41], RF [42], Case Based Reasoning [30], DT [25], ensemble methods [16,28,29,43,44], EM [31], Fuzzy clustering [33], K-means clustering [32], Association Rule Mining [45], and the Artificial Immune Systems [46,47].

In these techniques, researchers have applied several statistical and machine learning techniques to predict fault proneness models and reduce software development and maintenance costs. Among them, the machine learning technique is the most popular [1]. The majority of software defect prediction

techniques build models using metrics and faulty data from an earlier deployment or identical objects and then use the models to predict whether the modules presently under development contain defects, which is called a supervised learning approach [7]. Among the supervised learning techniques, ANN is one of the most popular, having received a great deal of attention) [39,48,49]. It should be pointed out that the ANN technique has some drawbacks in application for software defect prediction, the most important being the difficulty in determining the best neural network architecture in each application domain [49]. In contrast, there are other approaches, for example, clustering [33], which do not use previous data; these approaches are called unsupervised learning approaches. It is worth pointing out that some researchers, for example [50], classify software defect prediction techniques into descriptive and predictive techniques.

The usage of machine learning algorithms has increased in the last decade and is still one of the most popular methods for defect prediction [51,52]. Challagulla et al. [53] conducted an empirical assessment to evaluate the performance of various machine learning techniques and statistical models for predicting software quality. The experiments on four different real-time software defect datasets using different predictor models revealed that the 1R rule-based classification learning algorithm and Instance-based learning along with Consistency-based subset evaluation technique is more consistent in achieving accurate predictions as compared with other models. Based on their results, the authors presented a high-level design of an intelligent software defect analysis tool for defect assessment and dynamic monitoring of software modules. Catal and Diri [54] investigated the effects of data size, metrics, and feature selection techniques on software defect prediction. Nine classifiers were examined to explore which classifier performs best before and after applying feature reduction. They showed that NB is the best prediction algorithm for small datasets while Random Forests gives the best prediction performance for large datasets. Kaur and Pallavi [55] discussed the utilization of numerous machine learning approaches—for example, association mining, classification, and clustering in software defect prediction—but did not provide a comparative performance analysis of the techniques. Kumar and Gopal [56] proposed a binary classifier referred as LSTSVM which is the Least Square variant of Twin Support Vector Machine. The experiments showed that LSTSVM has comparable classification accuracy to Twin Support Vector Machine (TSVM) but with considerably lesser computational time. Agrawal and Tumar [57] proposed a feature selection based on the LSTSVM model for software defect prediction. A comparative analysis of various classification approaches against four PROMISE datasets showed the superiority of the proposed predictive model over other models, i.e., SVM and DT, in three datasets. Again, Tumar and Agrawal [58] developed a software defect prediction system using a weighted LSTSVM to consider misclassification cost of defective software modules. A comparison has been performed between the proposed approach and nine of the existing approaches using different performance measures. The results on eight datasets demonstrated the effectiveness of the proposed approach. Shukla and Verma [59] reviewed and analysed various literature studies on defect prediction area, investigated recent advancement in this area, and drew various conclusions. Dwivedi and Singh [60] analysed and compared various data mining classification and prediction techniques such as NN, NB, and *k*-NN for the software defect prediction models. The results showed that NN can outperform other two classifiers with the average accuracy of 91.54%. Wang et al. [8] proposed to leverage the directly learned semantic features to build machine learning models for predicting defects. The results on ten open source projects showed that the automatically learned semantic features using Deep Belief Network (DBN) improved within-project defect prediction on average by 14.7% in precision, 11.5% in recall, and 14.2% in F1. To reduce the complexity of metric selection and defect prediction, Huda et al. [61] proposed a framework for finding significant metrics to build and evaluate an automated software defect prediction model, using a hybrid combination of wrapper and filter techniques. Experimental results with eight NASA software datasets showed that the proposed hybrid approaches can select the most significant metrics with high prediction accuracy compared with conventional wrapper or filter approaches in some of the datasets. The highest accuracy achieved by the hybrid approach was almost 91% at different subset of metrics. Recently, Bowes et al. [62]

performed a sensitivity analysis for the prediction uncertainty produced by four different classifiers. Their results showed that classifier ensembles with decision-making strategies that are not based on majority voting are likely to perform best. Zhou et al. [38] proposed a new deep forest model to build the defect prediction model (DPDF). Their results on 25 open source projects from four public datasets showed that the DPDF increased AUC value by 5% compared best traditional machine learning algorithms.

3. Preliminaries

In this section, we briefly describe each of the algorithms utilized in the proposed approach.

3.1. NB

NB is a statistical probability-based classifier based on the Bayes theorem. NB is a family of algorithms based on a common principle, which assumes that all of the predictors are equally important and independent of each other [63]. In other words, when the class variable is given, it assumes the presence or absence of a particular feature is not related to the presence or absence of any other feature [64]. Instead of simple classification, NB reports the probability of an instance belonging to each individual class. In our case, the class with the highest posterior probability is the outcome of prediction that predicts whether a software module is defective or non-defective.

3.2. k-NN

k-NN is an instance-based learning method that classifies instances within a dataset by assigning the label of the closest neighbour to each new pattern during the testing phases. If the instances are tagged with a classification label, then the majority class of the closest *k* neighbours is assigned to the unclassified instance. Although the power of *k*-NN has been proven in a number of real domains, they have large storage requirements and their performance is sensitive to the choice of the *k*.

3.3. DT

DT is a logic-based learning method that classifies instances by sorting them based on feature values. The main idea underlying DT for classification tasks is the recursive partition of the data space; thus, a DT can be equivalently expressed as a set of rules. DT utilizes a tree-like data structure where each node in the tree represents a feature in an instance to be classified, whereas each branch represents a value that the node can assume [65]. The classification of instances starts at the root node, and instances are sorted based on their feature values. The most well-known algorithm in the literature for building tree is the C4.5, which is an extension of the ID3 algorithm. Although DT can effectively deal with nonlinear relationships, it is sensitive to noisy data and also may lead to overfitting.

3.4. Adaboost

AdaBoost is a widely used boosting algorithm that constructs an ensemble by performing multiple iterations each time with different instance weights and adjusts adaptively to the errors returned by classifiers from previous iterations [66,67]. Changing the weights of training instances in each iteration forces the learning algorithms to put more emphasis on instances that were incorrectly classified previously and less emphasis on instances that were correctly classified previously. In other words, weights of misclassified instances are increased, whereas weights of correctly classified instances are decreased. This will ensure misclassification errors for these misclassified instances count more heavily in the next iterations. AdaBoost uses the predictions of multiple weak classifiers and gives a final prediction through combined voting on techniques. Weak classifiers as originally defined by Freund and Schapire are classifiers that perform a little better than random guessing [68].

3.5. Bagging

Bagging is an ensemble technique that is used to improve the stability and accuracy of machine learning algorithms by combining the prediction of multiple weak classifiers [69]. Bagging works better for unstable learning algorithms where a little change in the training set results in large changes in predictions (i.e., ANN, DT). Bagging predicts an outcome several times from different training sets that are combined either by voting or with uniform averaging [70]. To describe the bagging algorithm, consider a dataset with N instances and a binary class label. The following steps summarize the Bagging algorithm:

1. Generate a random training set of size N with replacement from the data.
2. Train the random training set using any classification technique.
3. Assign a class to each node.
4. Repeat steps 1 to 3 many times.
5. Use voting to predict the class label.

3.6. RF

The RF classifier is a special case of Bagging consisting of a collection of tree-structured classifiers. RF selects random features in order to create bootstrap models using decision trees [71]. To do so, it creates a random forest of multiple decision trees by selecting data and variables randomly. A subset of instances is chosen randomly from the selected attributes and assigned to the learning algorithm. The forest selects the classification that has the most votes over all the trees in the forest. RF relies on aggregating the output from many "shallow" trees (called stumps), which are tuned and pruned without much analysis, so that the errors from many stumps will disappear when aggregated and lead to a more accurate prediction. Randomization in RF appears in two places:

1. Each tree is trained using a random sample with replacement from a training set.
2. When training individual trees, a random subset of features is used for searching for splits. The randomization reduces the correlations among trees, which improves the predictive performance.

3.7. XGB

XGB is a decision-tree-based supervised learning algorithm that implements a process called Gradient Boosting to construct an ensemble learner [72]. XGB optimises a collection of weak decision tree learning models to build an accurate and reliable predictor, decision tree ensemble, which uses the output of the weak learners in the final prediction. XGB improves upon the base Gradient Boosting Machines (GBMs) framework through algorithmic enhancements (i.e., Regularization, Sparsity Awareness, Weighted Quantile Sketch) and software and hardware optimization techniques (i.e., Parallelization, Tree Pruning). These improvements yield superior results using less computing resources in the shortest amount of time.

3.8. K-Means Clustering

K-means clustering is one of the most popular unsupervised learning methods. The main goal of K-means is to group similar data instances together and find patterns in the given datasets. To achieve this goal, K-means defines number of clusters (K) and then groups the similar elements into these clusters. It starts by selecting the centriods, which are the starting points of the clusters. In the next step it assigns the instances to the closest centroid and then updates the positions of the centroids iteratively until the centroids are stabilized or the predefined maximum number of iterations is reached.

Given a dataset of n instances $S = \{x_1, \ldots, x_n\} \in R^d$, and an integer number K, K-means algorithm aims to find $C = \{c_1, \ldots, c_K\}$, the set of centroids with respect to the following error function:

$$E(C) = \sum_{x \in S} \min_{i=1,\ldots,K} \|x - c_i\|^2 \quad (1)$$

As mentioned before, K-means assigns instances to one of the specified clusters according to the similarity between them. To measure the similarity, it usually uses the Euclidean distance between the instance and the centroids.

4. Proposed Approach

In this paper, we propose an approach for software defect classification where models are developed based on clustered patterns. The approach is composed of three main phases: In the first phase, a clustering process is applied on the training data to segment it into a set of groups of similar instances. In the second phase, different classifiers are trained based on the generated groups from the first phase. The third phase evaluates the developed models and uses them for predicting unrepresented instances. These three phases are illustrated in Figure 1 and described in details in the following three subsections.

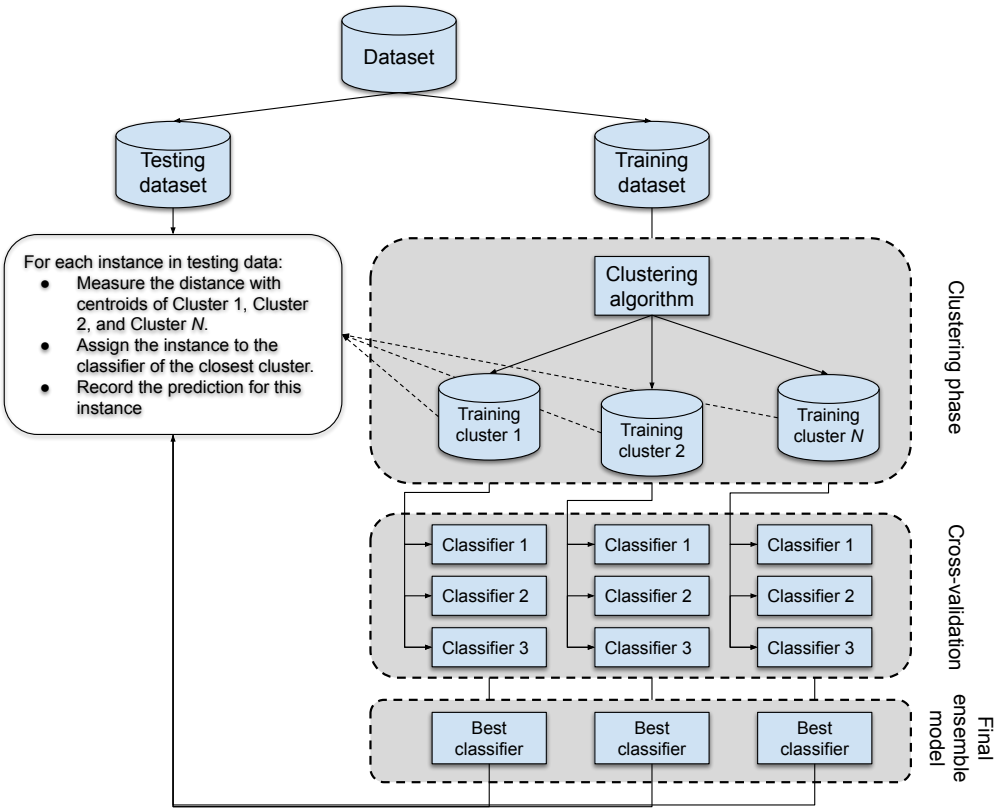

Figure 1. Proposed methodology for software defect prediction.

4.1. Clustering Phase

The clustering phase is the first phase of the hybrid algorithm. The idea is to start a preprocessing step to prepare the data for developing the classification algorithms. The data is split into two parts: the training part, which is the only part that is used in this stage for clustering, and a testing part, which is used to evaluate the performance of the trained models. During this preprocessing step, we start by clustering training data into a set of predefined number of clusters. We can use any

clustering technique, but in our work one of the most popular clustering techniques, the k-means algorithm, will be used.

4.2. Models Development Phase

After segmenting the training data into a set of clusters, the next step is to develop a classification model for each cluster. To do that, several classification algorithms are trained and evaluated on each cluster using the cross-validation methodology. The goal is to find the most suitable and expert model for each cluster. For example, suppose we have three classification algorithms called X, Y, and Z. All algorithms will be trained and evaluated based on each cluster. For example, if algorithm Y produced the highest average accuracy over the cross-validation process based on a given cluster, then Y will be assigned to this cluster for future predictions because it showed higher prediction power than algorithms X and Z. It is important to note that when there is a cluster of only one class the classifier works as one class classification algorithm. So, it trains based on one class in the training phase and detects the other class in the testing phase as outlier. After finishing this phase, each cluster will have its own expert model. Note that the best classifier can be different from one cluster to another. Figure 2 gives an example of this phase with three different classifiers trained on three clusters, and it shows how the classifiers in the final model are selected.

In this work, two types of classifiers are implemented to produce two versions of the proposed approach. In the first version, basic classifiers are used. This version will be referred to as K-Means/Basic classifiers (KMB). In the second version, ensemble classifiers will be utilized. The latter version will be referred to as K-Means/Ensemble classifiers (KME).

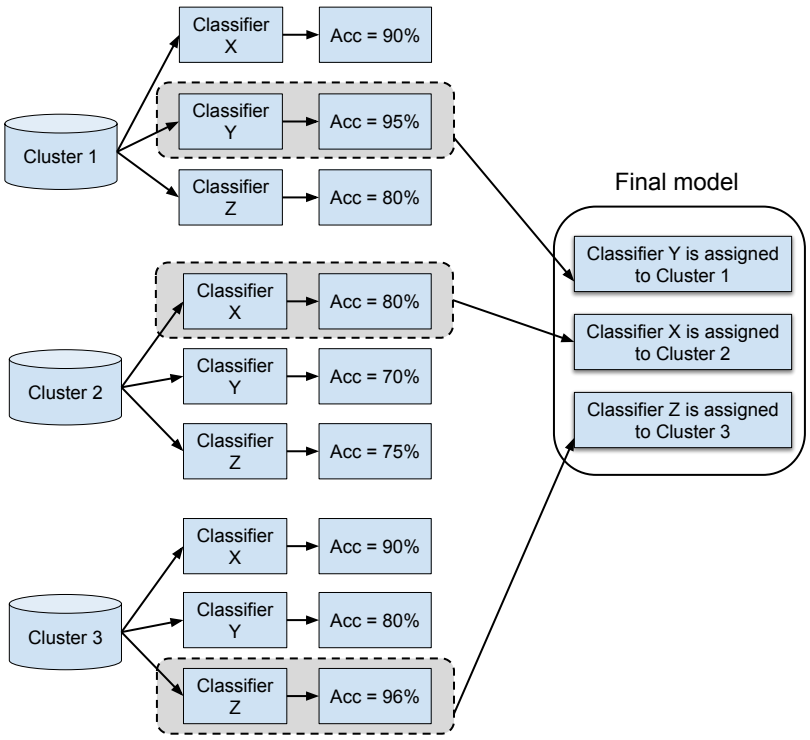

Figure 2. Example of the model development phase of the proposed approach.

4.3. Testing Phase

In the testing phase, we are concerned with the testing data generated in the first phase. For each instance in the testing data, we must specify to which cluster it belongs by calculating the distance between the instance and each centroid of the clusters. As a result, the instance will belong to the closest cluster (most similar), and it will be given to the model that has been assigned to the cluster

in the training phase for final prediction. To determine the similarity, we use the Euclidean distance between the testing instance I and the centroid C, which can be defined as follows:

$$d(I, C) = \sqrt{\sum_{i=1}^{d} (I_i - C_i)^2} \tag{2}$$

where d is the number of input features in the dataset.

After classifying all instances in the testing data, we can use the predictions against the actual values of classes to evaluate the performance of the given hybrid algorithm. The procedure of the algorithm is explained in Algorithm 1.

Algorithm 1: Ensemble with clustering.

input : D = $\{x_1, x_2, ..., x_n\}$ //Training dataset, D, which contains a set of training instances and their associated class labels.

```
Select F classifiers {f_1, f_2, ..., f_i}
Set k  //k number of clusters.
Split data into training data (λ) and testing data (Γ)
Cluster λ into k clusters = > {C_1, C_2, ..., C_k}
//Training phase.
foreach classifier f_i in F do
    for j ← 1 to k do
        Split C_j to cv folds//inner cross-validation
        cvError = 0
        for l ← 1 to cv  do
            Train f_i on cv − 1 folds
            compute error E_l on the held out fold
            cvError = cvError+E_l
    avgError_i = cvError/cv
For each C_j select f_i with the lowest avgError
//Testing phase.
foreach instance I in Γ do
    Calculate the distance between I and each C_j
    Find the closest C_j to I and its corresponding f_i
    Prediction [I] = f_i(I)
```

5. Model Evaluation Metrics

To evaluate the proposed software defect prediction model, we refer to the confusion matrix shown in Table 1, which is the primary source for accuracy estimation in classification problems. Based on this confusion matrix, the following criteria are used for evaluation:

1. Recall: is the fraction of relevant instances that have been retrieved over the total amount of relevant instances (i.e., coverage rate). It can be expressed by the following equation:

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

2. Precision: is the ratio of relevant instances among the retrieved instances. It can be given by the following equation:

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

3. G-mean: is the geometric mean of the recalls of each class and it can be measured by the following equation:

$$G - mean = \sqrt{\frac{TP}{TP + FN} \times \frac{TN}{TN + FP}} \quad (5)$$

Table 1. Confusion matrix.

	Actual	
	Defect	**No Defect**
Predicted defects	TP	FP
Predicted non-defects	FN	TN

6. Datasets Description

To facilitate the replication and verification of our experiments, the proposed approach is applied to a series of 21 well-studied public and available online software defect benchmark datasets with various attributes and instances. Eleven of the studied datasets are obtained from the NASA corpus while ten from the PROMISE software engineering corpus [73]. However, the NASA corpus is a known-to-be noisy corpus [74,75]. To avoid the effect of such noisy data on the results of our experiments, we use the cleaned version of the NASA corpus as provided by [74], which is available online (https://figshare.com/articles/MDP_data_sets_D_and_D_-_zipped_up/6071675). The NASA datasets were collected from real software projects from different domains by NASA and have various software modules developed in several different programming languages including, C, C++, and Java, various scales of lines of code, and various types of software metrics. For instance, in the cleaned version of the NASA corpus, the JM1 dataset consists of 7782 instances (1672 defective/6110 defect-free) where each instance includes a total of 22 attributes, of which five are different lines of code measures, three are McCabe metrics, four are base Halstead measures, eight are derived Halstead measures, one is a branch-count, and one is a decision attribute that indicates whether a particular instance is defective or non-defective. The PROMISE datasets were collected from open source software projects developed in a variety of settings (e.g., Apache, GNU) which provides different metrics than the NASA corpus does. Table 2 shows information and some general statistics of each dataset.

Table 2. Datasets specifications.

Datasets	Attributes	Instances	Defects	Non-Defects	Defects%	Non-Defects %
cm1	38	327	42	285	12.8	87.2
jm1	22	7782	1672	6110	21.5	78.5
kc1	22	1183	314	869	26.5	73.5
kc3	40	194	36	158	18.6	81.4
mc1	39	1988	46	1942	2.3	97.7
mw1	38	253	27	226	10.7	89.3
pc1	38	705	61	644	8.7	91.3
pc2	37	745	16	729	2.1	97.9
pc3	38	1077	134	943	12.4	87.6
pc4	38	1287	177	1110	13.8	86.2
pc5	39	1711	471	1240	27.5	72.5
ant-1.7	21	745	166	579	22.3	77.7
camel-1.6	21	965	188	777	19.5	80.5
ivy-2.0	21	352	40	312	11.4	88.6
jedit-4.3	21	492	11	481	2.2	97.8
log4j-1.2	21	205	189	16	92.2	7.8
lucene-2.4	21	340	203	137	59.7	40.3
poi-3.0	21	442	281	161	63.6	36.4
tomcat-6	20	858	77	781	9	91
xalan-2.6	21	885	411	474	46.4	53.6
xerces-1.4	21	588	437	151	74.3	25.7

7. Experiments and Results

The experiments was executed 30 independent time then the average of the results was calculated. The experiments will be conducted in three steps:

- The best number of clusters is experimented for each dataset, and the best model for each cluster is found.
- The proposed approach is experimented based on utilizing simple and common classifiers (NB, *k*-NN, and DT).
- The proposed approach is experimented based on utilizing powerful ensemble classifiers (Bagging, AdaBoost, RF, and XGB).

For *k*-NN, the number of neighbours is set to 3 as this value showed best performance based on the training data compared to other values (i.e., 1, 5, 7, and 9). For the ensemble classifiers, the selected base classifier is decision tree, and the ensemble size is set to 100. The latter settings yield the best performance based on the training data with the least computation effort.

7.1. Finding the Best Number of Clusters and Their Corresponding Models

The number of clusters in our final model will be determined based on the G-mean results of the training phase. Cross-validation approaches can help avoid overfitting in model selection [76]. Therefore, the training process is conducted based on two-folds cross-validation to avoid overfitting. Four settings are experimented to determine the required number of clusters: 3, 5, 7 or 9. The number of clusters that yield the highest G-mean value will be selected to be applied for the final model in the testing phase. Tables 3 and 4 show the G-mean results of the cross-validation training phase of our approach based on the basic classifiers and ensemble ones, respectively. As it can be seen in the tables, the best number of clusters varies from one dataset to another. This confirms that the software defect benchmark datasets are varied in their nature, where different groups can be identified, and these groups have a number of similar patterns. Tables 3 and 4 also show the distribution of classes in each cluster. It is important to note here that sometimes the clustering process produce clusters that contain only class. For such cases, any new instance that is closer to the center of a one-class cluster will be simply given the same class of the cluster.

For each cluster that results from the previous step, the best performing classifier is assigned. Demonstrating the best performing models for KMB and KME, Figures 3 and 4 show the frequency of the best models over all datasets. In the case of KMB, we can see that the DT is the most frequent classifier in most of cases, followed by NB and *k*-NN. In the case of KME, Bagging is the most frequent model, followed by XGB, RF and AdaBoost, respectively. This supports the idea that there is no dominating classifier for all the data patterns, and each group of similar patterns needs a model that fits its particular characteristics.

Table 3. G-mean results of two-folds cross-validation for the proposed approach based on the basic classifiers.

Datasets	KMB					Best # of Clusters	Distribution of Classes Defects:Non-Defects
	1C	3C	5C	7C	9C		
cm1	0.518	0.608	0.723	0.744	**0.845**	9	7.7:92.3, 12.8:87.2, 14.3:85.7, 22.2:77.8, 50:50, 0:100, 0:100, 66.7:33.3, 13.8:86.2
jm1	0.615	0.622	**0.631**	0.625	0.621	5	16.6:83.4, 23.2:76.8, 49.6:50.4, 12.2:87.8, 100:0
kc1	0.628	0.594	**0.722**	0.625	0.592	5	17.8:82.2, 69.4:30.6, 50:50, 15:85, 44.2:55.8
kc3	**0.685**	0.546	0.441	0.423	0.464	1	21.6:78.4
mc1	0.385	0.559	0.340	0.680	**0.716**	9	5.6:94.4, 0:100, 3.5:96.5, 0:100, 2.6:97.4, 2:98, 0:100, 0:100, 50:50
mw1	**0.593**	0.519	0.532	0.517	0.526	1	7.9:92.1
pc1	0.556	**0.603**	0.548	0.414	0.453	3	13.5:86.5, 11.7:88.3, 2.3:97.7
pc2	0.658	0.575	**0.897**	0.658	0.837	5	0:100, 0.8:99.2, 6.3:93.7, 0:100, 5.4:94.6
pc3	**0.633**	0.549	0.565	0.534	0.538	1	11.9:88.1
pc4	0.634	0.606	**0.691**	0.471	0.234	5	5.3:94.7, 5.8:94.2, 9.8:90.2, 12.1:87.9, 31.4:68.6
pc5	**0.671**	0.641	0.617	0.653	0.559	1	26.1:73.9
ant-1.7	0.574	0.460	0.427	0.486	**0.639**	9	2.8:97.2, 43.1:56.9, 5.8:94.2, 12.7:87.3, 86.7:13.3, 15:85, 16.7:83.3, 13.1:86.9, 40:60
camel-1.6	**0.619**	0.588	0.611	0.540	0.562	1	17.3:65.4
ivy-2.0	0.589	**0.671**	0.633	0.630	0.564	3	13.8:86.3, 5.6:94.4, 7.1:92.9
jedit-4.3	**0.826**	0.458	0.458	0.633	0.650	1	0.8:99.2
log4j-1.2	0.773	0.500	0.874	0.975	**0.989**	9	80:20, 93.1:6.9, 100:0, 87:13, 50:50, 100:0, 100:0, 100:0, 100:0
lucene-2.4	0.734	**0.822**	0.720	0.592	0.548	3	51.9:48.1, 40.4:59.6, 67.8:32.2
poi-3.0	**0.817**	0.653	0.526	0.551	0.567	1	60.2:39.8
tomcat-6.0	0.640	0.666	**0.726**	0.346	0.513	5	37:63, 8.6:91.4, 2.3:97.7, 5.4:94.6, 9.6:90.4
xalan-2.6	**0.756**	0.635	0.723	0.710	0.644	1	45.9:54.1
xerces-1.4	0.843	0.864	0.843	**0.886**	0.759	7	89.3:10.7, 50:50, 92.1:7.9, 48.3:51.7, 90.5:9.5, 78.6:21.4, 100:0

Table 4. G-mean results of two-folds cross-validation for the proposed approach based on the ensemble classifiers.

Datasets	KMB					Best # of Clusters	Distribution of Classes Defects : Non-Defects
	1C	3C	5C	7C	9C		
cm1	0.417	0.653	0.758	0.748	**0.845**	9	21.4:78.6, 5.9:94.1, 50:50, 0:100, 0:100, 41.7:58.3, 0:100, 13:87, 40:60
jm1	0.505	0.585	0.596	**0.612**	0.606	7	19.7:80.3, 18.6:81.4, 50.6:49.4, 11.3:88.7, 100:0, 34.7:65.3, 68.2:31.8
kc1	0.562	0.597	**0.746**	0.633	0.623	5	13.2:86.8, 42.9:57.1, 58.3:41.7, 26.2:73.8, 21.1:78.9
kc3	**0.759**	0.569	0.481	0.447	0.491	1	25.8:74.2
mc1	0.373	0.504	0.315	0.696	**0.718**	9	1.2:98.8, 1.6:98.4, 8.2:91.8, 0.9:99.1, 0:100, 0:100, 1.2:98.8, 13.6:86.4, 0:100
mw1	**0.661**	0.526	0.539	0.522	0.524	1	8.7:91.3
pc1	0.565	**0.642**	0.497	0.429	0.484	3	21.6:78.4, 11.2:88.8, 7.9:92.1
pc2	0.402	0.484	**0.913**	0.672	0.840	5	0:100, 1.7:98.3, 2.3:97.7, 4.5:95.5, 16.7:83.3
pc3	**0.671**	0.492	0.665	0.573	0.589	1	13.6:86.4
pc4	0.695	0.664	**0.697**	0.503	0.229	5	10.1:89.9, 25.9:74.1, 2.6:97.4, 1:99, 19.6:80.4
pc5	0.688	0.690	0.564	**0.701**	0.577	7	38.5:61.5, 21.8:78.2, 29.1:70.9, 36.7:63.3, 24.4:75.6, 72.7:27.3, 16.8:83.2
ant-1.7	0.603	0.554	0.461	0.536	**0.726**	9	26.3:73.7, 35:65, 8.3:91.7, 11.9:88.1, 4.8:95.2, 10.3:89.7, 92.3:7.7, 57.1:42.9, 21.9:78.1
camel-1.6	0.500	0.496	**0.607**	0.524	0.594	5	11.5:88.5, 20.1:79.9, 3.8:96.2, 27.6:72.4, 21.6:78.4
ivy-2.0	0.657	**0.861**	0.719	0.681	0.636	3	15.4:84.6, 0:100, 18.7:81.3
jedit-4.3	**0.716**	0.323	0.475	0.608	0.627	1	1.6:98.4
log4j-1.2	0.667	0.472	0.890	0.977	**0.987**	9	91.7:8.3, 100:0, 100:0, 100:0, 100:0, 100:0, 100:0, 85.7:14.3, 90.5:9.5
lucene-2.4	0.809	**0.869**	0.781	0.633	0.591	3	42.5:57.5, 66:34, 51.1:48.9
poi-3.0	**0.838**	0.741	0.625	0.635	0.639	1	61.1:38.9
tomcat-6.0	0.651	0.703	**0.706**	0.367	0.549	5	12.3:87.7, 5.1:94.9, 11.6:88.4, 0:100, 4.3:95.7
xalan-2.6	**0.793**	0.736	0.774	0.767	0.727	1	46.2:53.8
xerces-1.4	0.915	**0.969**	0.950	0.945	0.775	3	54.1:45.9, 93.2:6.8, 95.9:4.1

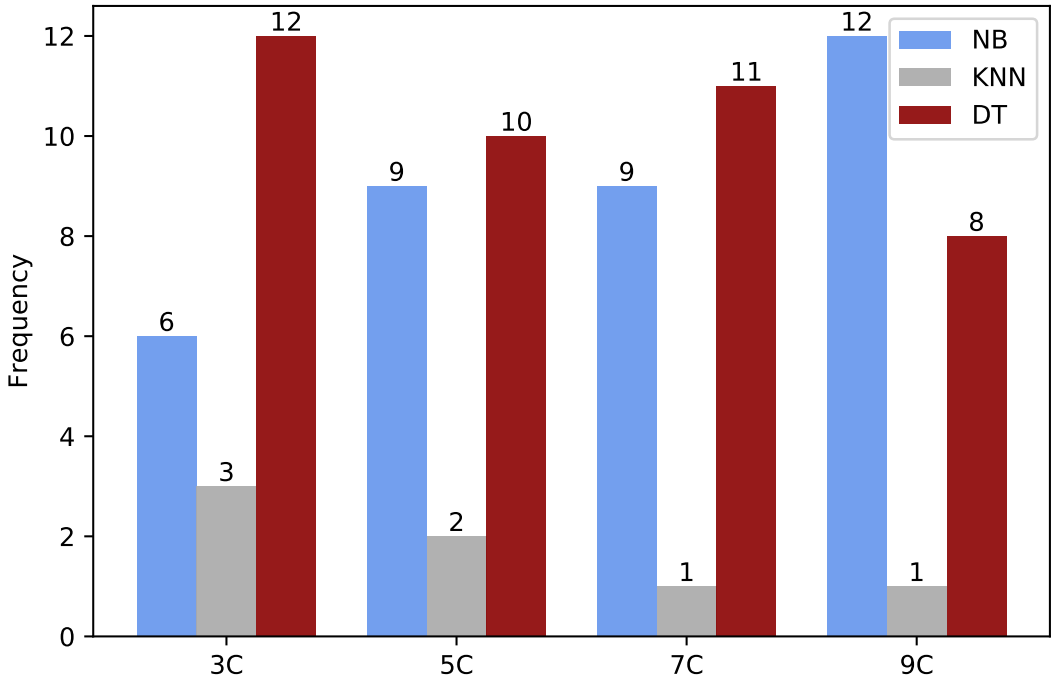

Figure 3. Frequency of the basic classifiers in K-Means/Basic classifiers (KMB) for different numbers of clusters.

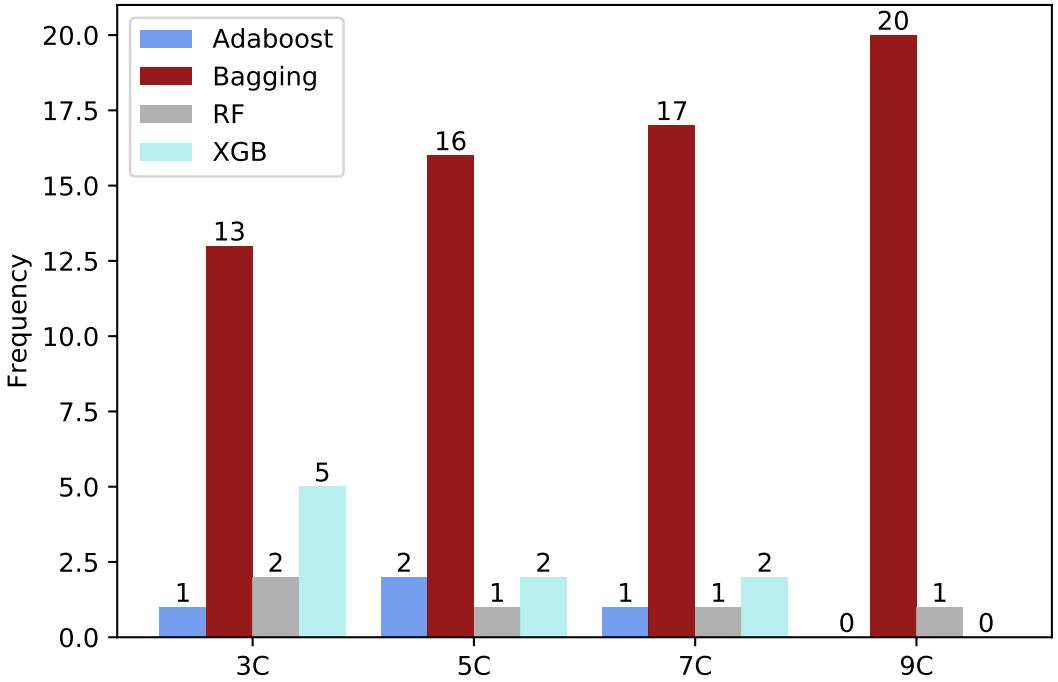

Figure 4. Frequency of the ensemble classifiers in K-Means/Ensemble classifiers (KME) for different numbers of clusters.

7.2. KMB vs. Basic Classifiers

In this part of the experiments we verify the performance of the KMB version of our proposed approach by experimenting it based on the 21 benchmark datasets and comparing it with the basic classifiers NB, *k*-NN, and DT.

Because all of the datasets are highly imbalanced, considering the accuracy ratio for evaluation is misleading. Therefore, other metrics (i.e., recall, precision, and G-mean) should be examined.

The results of precision, recall and G-mean values are shown in Table 5. According to the results, we can see that KMB hits the best or very competitive precision and recall values for most of the datasets. The results of the G-mean evaluation measure reveal that KMB has better performance in 21 datasets, followed by NB and *k*-NN respectively, where NB achieved best results only in two datasets, and *k*-NN in one dataset.

For better visualization of the results, radar Figures 5–7 are plotted for KMB and the basic classifiers.

Table 5. Precision, Recall, and G-mean for the basic classifiers and KMB.

Datasets	NB			*k*-NN			DT			KMB		
	Pre	Rec	G-Mean	Pre	Rec	G-Mean	Pre	Rec	G-Mean	Pre	Rec	G-Mean
cm1	0.404	0.312	0.525	0.406	0.300	0.517	0.379	0.285	0.501	**0.527**	**0.437**	**0.630**
jm1	0.504	0.207	0.442	0.511	0.209	0.444	0.510	0.210	0.445	**0.716**	**0.728**	**0.804**
kc1	0.592	0.389	0.591	0.601	0.395	0.596	0.596	0.388	0.590	**0.713**	**0.694**	**0.787**
kc3	0.482	0.581	0.690	0.480	0.562	0.681	0.479	0.578	0.688	**0.733**	**0.798**	**0.848**
mc1	0.029	**0.880**	0.594	0.029	0.870	0.596	0.030	0.877	0.598	**0.693**	0.660	**0.794**
mw1	0.081	0.338	0.480	0.085	0.433	0.540	0.083	0.417	0.529	**0.578**	**0.675**	**0.722**
pc1	0.250	0.329	0.540	0.275	0.356	0.564	0.266	0.338	0.550	**0.661**	**0.721**	**0.823**
pc2	0.093	**0.872**	**0.789**	0.089	0.861	0.779	0.088	0.844	0.772	**0.424**	0.516	0.555
pc3	0.144	0.908	0.406	0.146	**0.920**	0.409	0.144	0.907	0.405	**0.680**	0.720	**0.827**
pc4	0.270	0.723	0.727	0.281	0.754	0.745	0.282	0.749	0.745	**0.800**	**0.785**	**0.870**
pc5	0.535	0.213	0.443	0.537	0.217	0.448	0.535	0.215	0.446	**0.760**	**0.775**	**0.836**
ant-1.7	0.413	**0.787**	0.745	0.433	0.763	0.751	0.433	0.755	0.748	**0.708**	0.682	**0.795**
camel-1.6	0.315	0.205	0.428	0.330	0.212	0.436	0.333	0.218	0.442	**0.687**	**0.695**	**0.796**
ivy-2.0	0.593	**0.580**	**0.732**	0.580	0.557	0.716	0.611	0.574	0.731	**0.709**	0.572	0.684
jedit-4.3	0.300	0.244	0.269	0	0	0	0	0	0	**0.716**	**0.885**	**0.933**
log4j-1.2	0.961	0.962	0.749	0.963	0.970	**0.765**	0.962	0.970	0.756	**0.973**	**0.979**	0.600
lucene-2.4	0.799	0.600	0.695	0.800	0.589	0.690	0.786	0.593	0.685	**0.835**	**0.837**	**0.801**
poi-3.0	**0.915**	0.626	0.749	0.902	0.601	0.728	0.901	0.607	0.732	0.906	**0.917**	**0.867**
tomcat-6.0	0.436	0.418	0.620	0.461	0.438	0.637	0.456	0.431	0.631	**0.686**	**0.747**	**0.848**
xalan-2.6	0.864	0.513	0.688	**0.878**	0.527	0.700	0.876	0.512	0.690	0.851	**0.855**	**0.859**
xerces-1.4	0.934	0.698	0.783	0.942	0.632	0.756	0.943	0.626	0.754	**0.972**	**0.964**	**0.935**

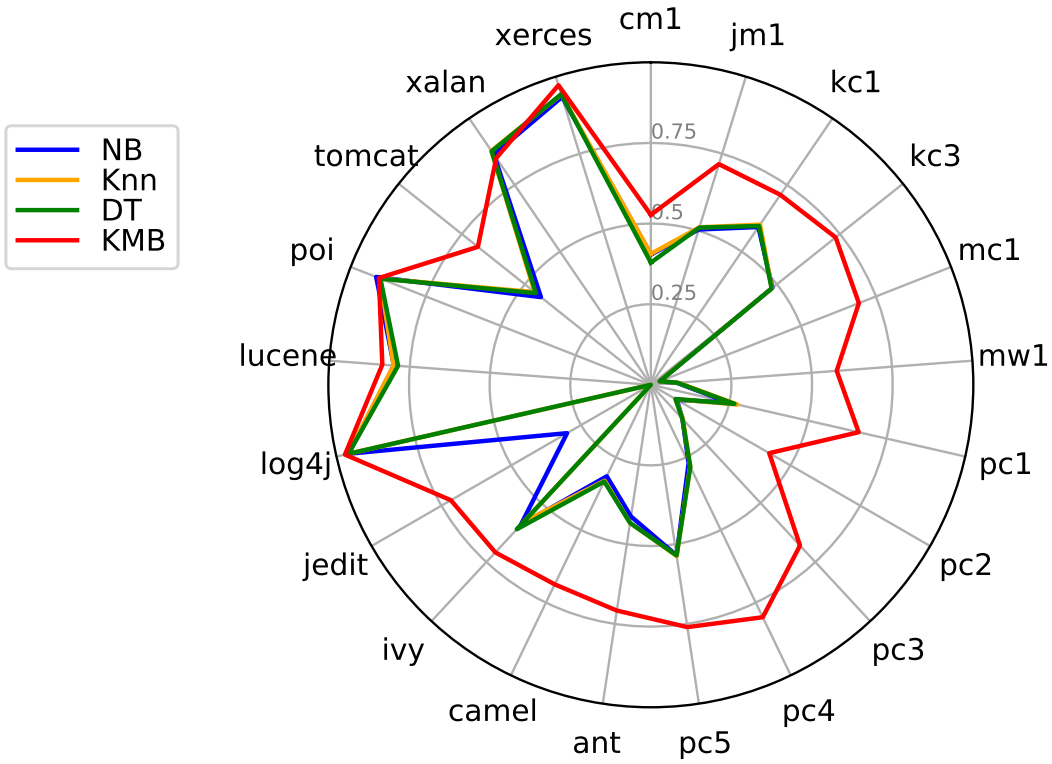

Figure 5. Precision for the basic classifiers and KMB.

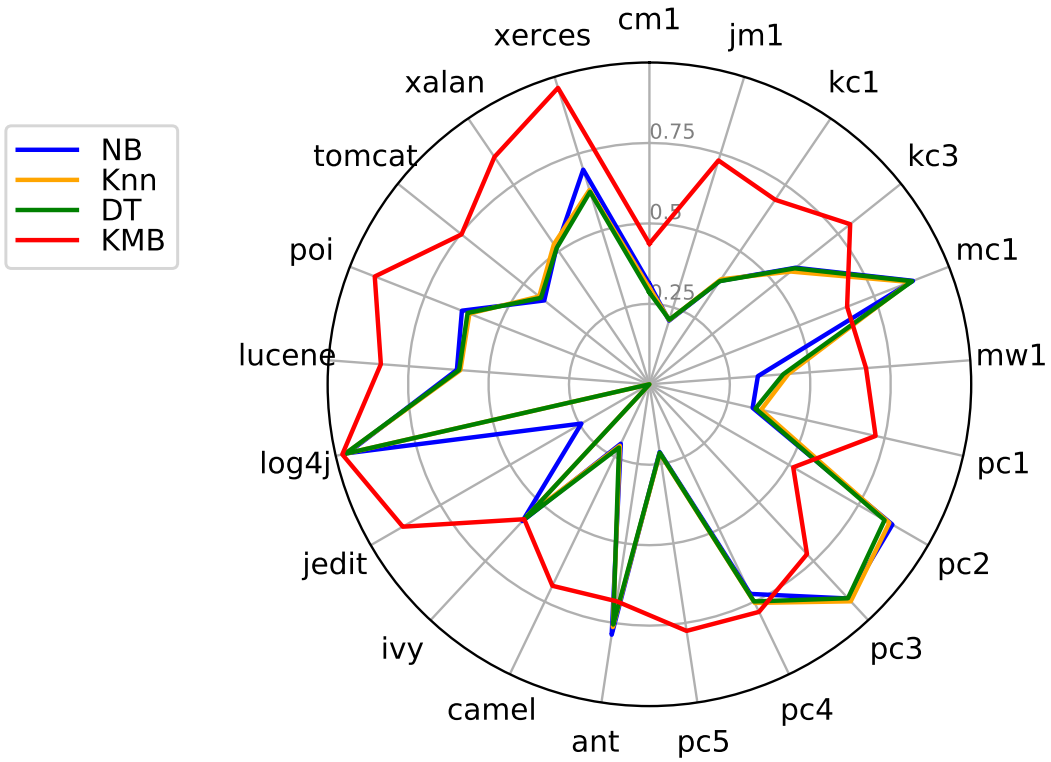

Figure 6. Recall for the basic classifiers and KMB.

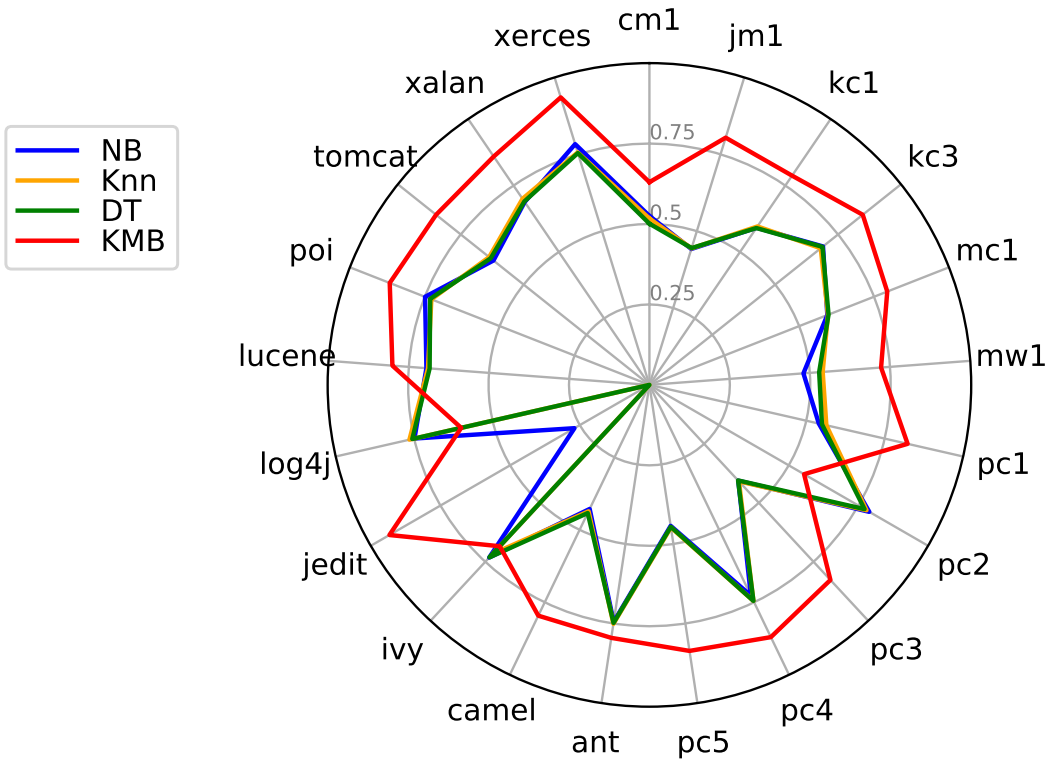

Figure 7. G-mean for the basic classifiers and KMB.

7.3. KME vs. Ensemble Classifiers

Here we experiment the KME version of the proposed algorithm, which combines ensemble classifiers instead of simple classifiers in an attempt to boost the predictive power of the approach.

For precision and recall, Table 6 shows that KME is dominating the top rates especially in regard to precision. Considering the G-mean results in Table 6, we can see that KME hits the best results in 19 datasets out of 21. For better visualization of the obtained results, radar Figures 8–10 are plotted for KME and the ensemble classifiers. By comparing the performance of KMB and KME in terms of G-mean results, Table 7 shows that the KMB achieves better results in 14 datasets out of 21 indicating that it is not necessary to apply the KME in all cases. This could be explained by the fact that KMB is a type of ensemble classification and it combines weak classifiers which could prevent overfitting. This is unlike the case of KME which combines powerful classifiers.

Table 6. Precision, Recall, and G-mean for the ensemble classifiers and KME.

Datasets	Bagging			AdaBoost			RF			XGB			KME		
	Pre	Rec	G-Mean	Pre	Rec	G-Mean	Pre	Rec	G-Mean	Pre	Rec	G-Mean	Pre	Rec	G-Mean
cm1	0.392	0.295	0.511	0.404	0.300	0.516	0.400	0.295	0.511	0.397	0.289	0.507	**0.783**	**0.623**	**0.775**
jm1	0.505	0.207	0.441	0.505	0.208	0.443	0.505	0.209	0.444	0.506	0.207	0.441	**0.846**	**0.669**	**0.780**
kc1	0.592	0.388	0.590	0.604	0.393	0.595	0.598	0.391	0.593	0.589	0.388	0.590	**0.808**	**0.663**	**0.758**
kc3	0.464	0.556	0.674	0.480	0.567	0.684	0.488	0.576	0.691	0.478	0.543	0.672	**0.832**	**0.792**	**0.862**
mc1	0.029	0.870	0.596	0.030	**0.892**	0.605	0.029	0.863	0.595	0.03	0.885	0.601	**0.866**	0.628	**0.761**
mw1	0.087	0.438	0.544	0.086	0.433	0.542	0.085	0.433	0.539	0.078	0.396	0.513	**0.729**	**0.519**	**0.594**
pc1	0.263	0.335	0.547	0.246	0.313	0.527	0.265	0.345	0.556	0.247	0.315	0.529	**0.828**	**0.659**	**0.803**
pc2	0.094	**0.886**	0.795	0.088	0.836	0.771	0.092	**0.886**	0.792	0.095	0.883	**0.796**	**0.901**	0.417	0.619
pc3	0.145	**0.916**	0.409	0.144	0.909	0.404	0.145	0.909	0.409	0.145	0.911	0.412	**0.776**	0.546	**0.726**
pc4	0.280	0.736	0.739	0.276	0.739	0.738	0.280	0.742	0.741	0.283	**0.749**	0.745	**0.881**	0.727	**0.840**
pc5	0.529	0.211	0.441	0.536	0.215	0.446	0.533	0.212	0.443	0.542	0.218	0.450	**0.832**	**0.730**	**0.811**
ant-1.7	0.438	**0.770**	0.756	0.434	0.756	0.749	0.434	0.768	0.753	0.436	0.760	0.752	**0.801**	0.635	**0.780**
camel-1.6	0.320	0.210	0.433	0.325	0.214	0.438	0.336	0.225	0.449	0.333	0.219	0.443	**0.841**	**0.621**	**0.746**
ivy-2.0	0.616	0.569	0.728	0.618	0.584	0.737	0.606	0.553	0.717	0.595	0.556	0.718	**0.858**	**0.703**	**0.828**
jedit-4.3	0	0	0	0	0	0	0	0	0	0	0	0	**0.919**	**0.885**	**0.938**
log4j-1.2	0.961	0.969	0.752	0.961	0.969	0.751	0.962	0.967	0.753	0.962	0.969	**0.761**	**0.969**	**1.000**	0.325
lucene-2.4	0.802	0.583	0.689	0.794	0.588	0.687	0.798	0.594	0.691	0.797	0.592	0.691	**0.862**	**0.853**	**0.831**
poi-3.0	0.906	0.593	0.726	0.896	0.592	0.721	0.904	0.602	0.73	0.906	0.603	0.732	**0.929**	**0.925**	**0.895**
tomcat-6.0	0.441	0.412	0.617	0.441	0.426	0.626	0.447	0.431	0.630	0.449	0.436	0.635	**0.788**	**0.567**	**0.743**
xalan-2.6	0.879	0.521	0.697	0.873	0.508	0.687	0.877	0.513	0.691	**0.880**	0.514	0.693	0.878	**0.820**	**0.857**
xerces-1.4	0.937	0.626	0.750	0.941	0.623	0.751	0.941	0.638	0.758	0.942	0.626	0.753	**0.976**	**0.982**	**0.923**

Table 7. G-mean for KMB and KME.

Datasets	G-Mean	
	KMB	KME
cm1	0.630	**0.775**
jm1	**0.804**	0.780
kc1	**0.787**	0.758
kc3	0.848	**0.862**
mc1	**0.794**	0.761
mw1	**0.722**	0.594
pc1	**0.823**	0.803
pc2	0.555	**0.619**
pc3	**0.827**	0.726
pc4	**0.870**	0.840
pc5	**0.836**	0.811
ant-1.7	**0.795**	0.780
camel-1.6	**0.796**	0.746
ivy-2.0	0.684	**0.828**
jedit-4.3	0.933	**0.938**
log4j-1.2	**0.600**	0.325
lucene-2.4	0.801	**0.831**
poi-3.0	0.867	**0.895**
tomcat-6.0	**0.848**	0.743
xalan-2.6	**0.859**	0.857
xerces-1.4	**0.935**	0.923

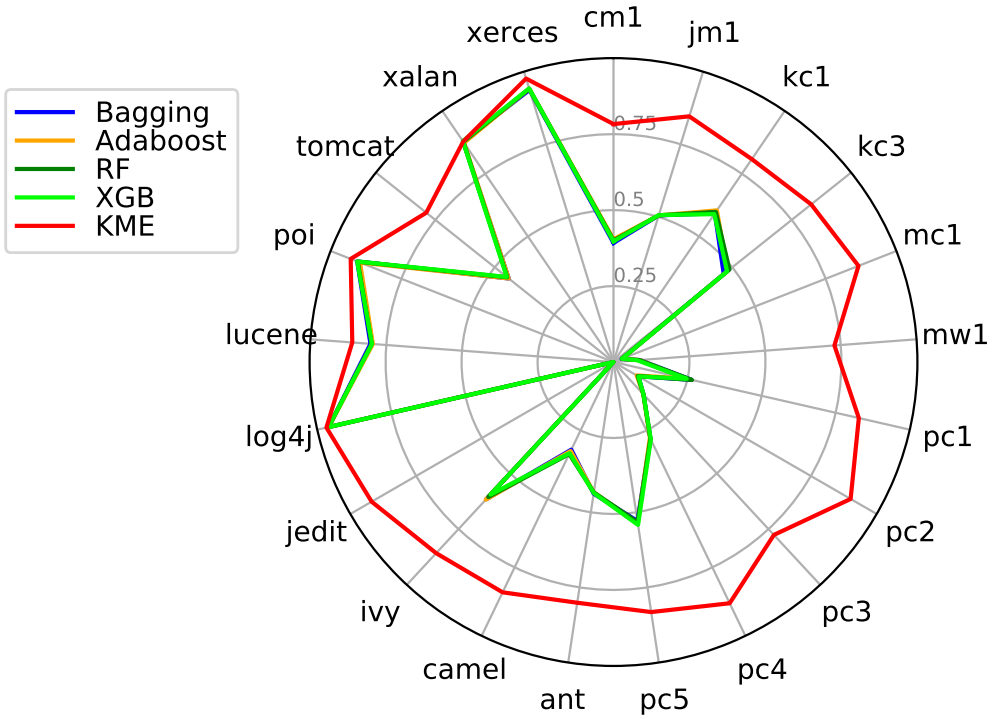

Figure 8. Precision for the ensemble classifiers and KME.

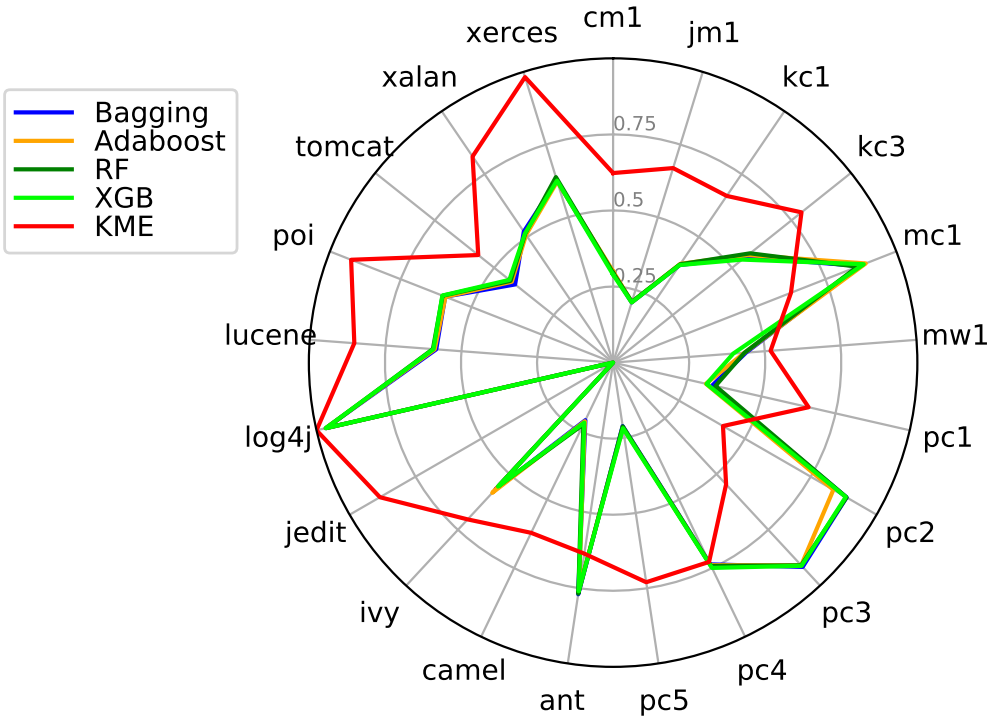

Figure 9. Recall for the ensemble classifiers and KME.

The boxplots for KMB and KME represented in Figures 11 and 12. These boxplots are created for reporting the G-mean of 30 independent runs for all datasets. The figures show that KMB and KME are very competitive and stable in most of the datasets. It can be noted that for few datasets KMB and KME exhibit more sensitive performance than the other datasets. Examples of these datasets are cm1, mc1, mw1 and pc2. This sensitivity can be due to the high imbalance ratio in these datasets, that is, misclassifying one instant from the rare class will highly impact the G-mean measure. For jedit-4.3

dataset, the single classifier approach failed to classify the rare instances therefore its recall is 0 and consequently the G-mean is 0. On the other hand, for the log4j-1.2 dataset, performance of of KMB and KME was worse than the other classifiers which could be reasoned to the clustering step which produced clusters that are harder for KMB and KME to cluster.

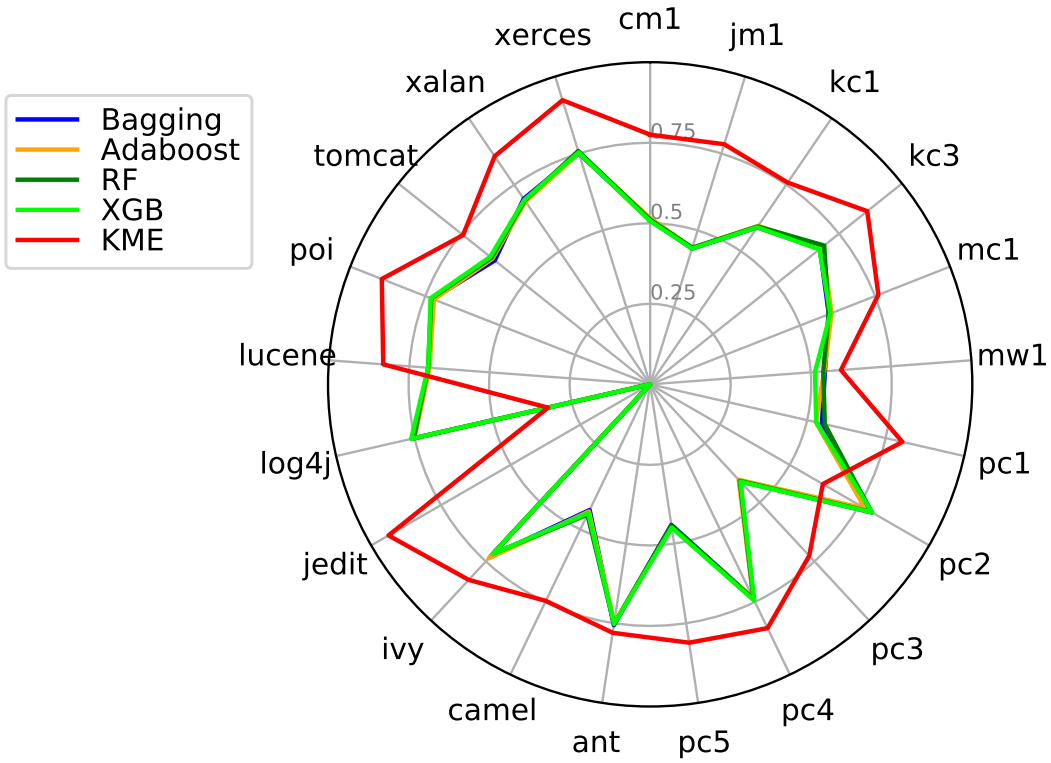

Figure 10. G-mean for the ensemble classifiers and KME.

Although the purposed approaches have been compared with the best traditional machine learning algorithms, we also compare the proposed KME with the results of one of the most recent sophisticated approaches called defect prediction based on deep forest (DPDF) [38]. Specifically, we used the common datasets used in their published results and in our approach, 13 datasets, to compare our approach against in terms of precision and recall as shown in Table 8. The results show that KME outperforms the DPDF in all datasets except for pc2 and tomcat-6.0.

Table 8. Comparison between defect prediction based on deep forest (DPDF) [38] and KME.

Datasets	Pre		Rec	
	DPDF	**KME**	**DPDF**	**KME**
mc1	0.290	**0.866**	0.020	**0.628**
mw1	0.630	**0.729**	0.430	**0.519**
pc1	0.250	**0.828**	0.130	**0.659**
pc2	**0.980**	0.901	**0.720**	0.417
pc3	0.260	**0.776**	0.070	**0.546**
pc4	0.770	**0.881**	0.210	**0.727**
pc5	0.610	**0.832**	0.370	**0.730**
ant-1.7	0.640	**0.801**	0.480	**0.635**
camel-1.6	0.460	**0.841**	0.120	**0.621**
lucene-2.4	0.690	**0.862**	0.820	**0.853**
poi-3.0	0.840	**0.929**	0.820	**0.925**
tomcat-6.0	**0.840**	0.788	0.120	**0.567**
xalan-2.6	0.760	**0.878**	0.680	**0.820**

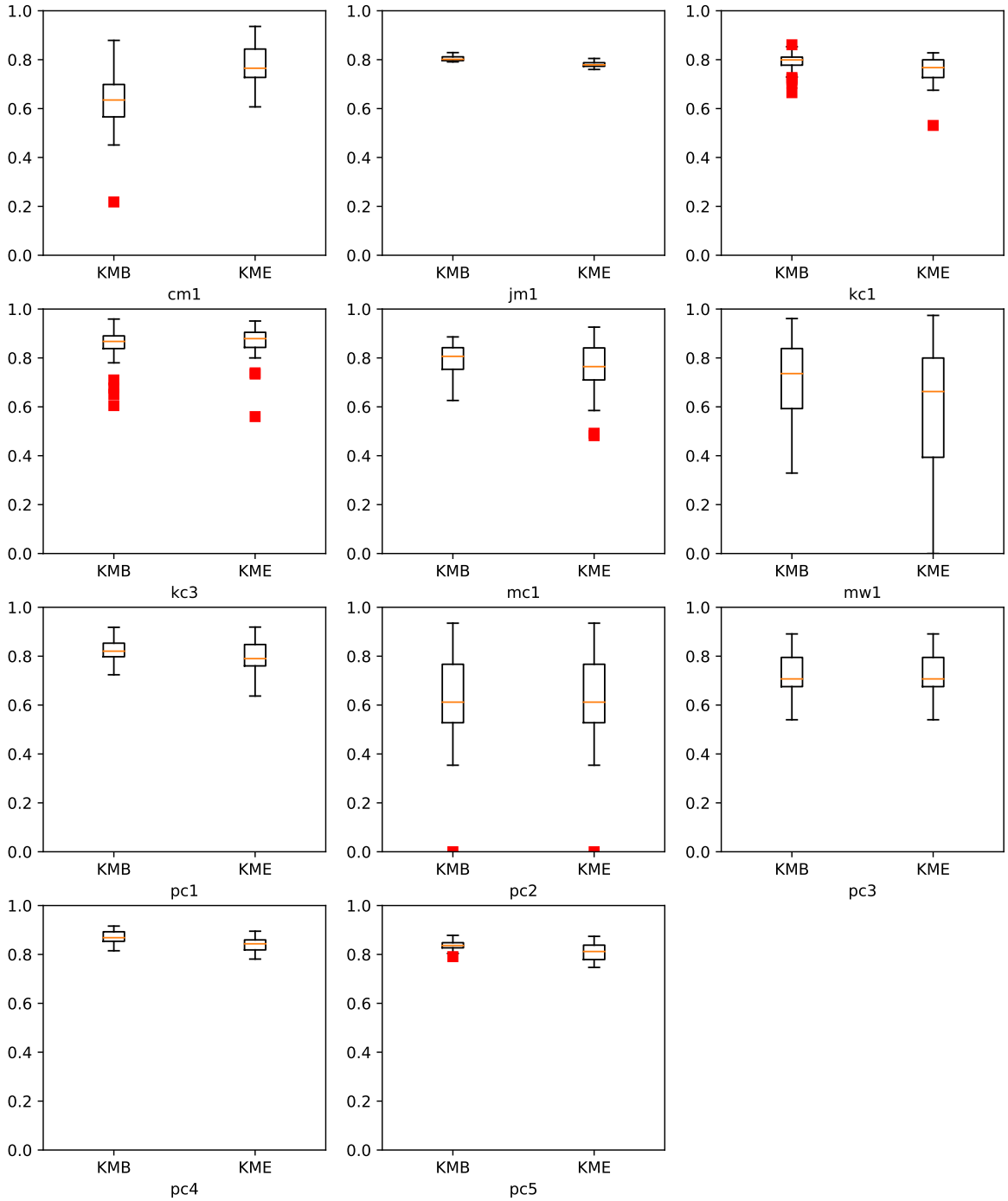

Figure 11. Boxplot representing the G-mean of KMB and KME for NASA datasets.

7.4. Statistical Test

A nonparametric statistical test—The Friedman test—of multiple group measures is usually used to approve the null hypothesis that the multiple group measures have the same variance using a precise level of significance. Alternatively, rejecting the null hypothesis approves that they have different variance values. We analyze the execution of the Algorithms using the Friedman test in SPSS and we run the test 21 times using different datasets. For each experiment we used H0 that there is no difference in the execution between the Algorithms and H1 that there is a difference in the execution of the Algorithms. We reject H0 for $p < \alpha$: as $\alpha = 0.05$ is used as the significance level in this hypothesis testing. The results of Friedman test are shown in Table 9. From the results, we can see that the Mean

ranks differ quite a lot in favor of the KME Algorithm and for the KMB Algorithm in almost all the experiments. The Chi-Square test statistics mainly summarize how differently our Algorithms were rated in a single number. The degrees of freedom in our experiments, 9 (Algorithms) variables − 1 = 8 degrees of freedom. The results show that there is a significant difference the execution of the Algorithms for all the experiments Since the p-value (Asymp. Sig.) < 0.05, and we cannot accept the null hypothesis of equal population distributions. Moreover, the whole table illustrates which Algorithm was ranked best versus worst. In other words, the Friedman test points out that our Algorithms were rated differently, Chi-Square values with a p-value ≤ 0.0000 for all experiments.

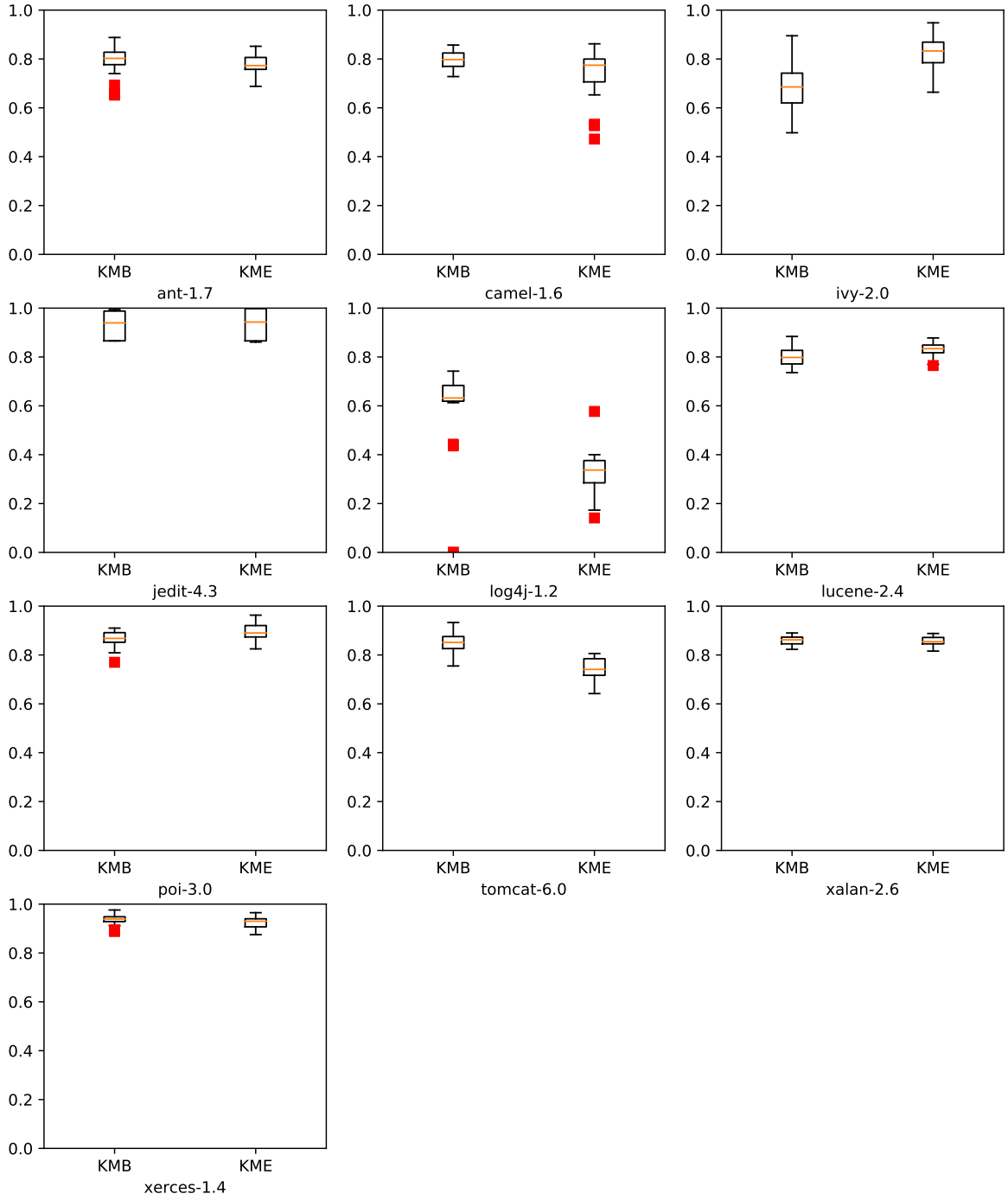

Figure 12. Boxplot representing the G-mean of KMB and KME for PROMISE datasets.

Table 9. Friedman test ranking.

	Mean Rank (df = 8, N = 30)										
	KME	**KMB**	**Ada**	**Bag**	**RF**	**XGB**	**NB**	**Knn**	**DT**	**Chi-Square**	**Asymp. Sig.**
cm1	**8.870**	7.000	4.700	4.000	4.130	3.850	4.450	4.330	3.670	98.860	0.000
jm1	8.000	**9.000**	4.100	3.570	4.300	3.770	4.030	3.930	4.300	129.751	0.000
kc1	8.100	**8.670**	4.130	3.530	4.000	3.900	4.080	4.650	3.930	121.136	0.000
kc3	**8.350**	7.920	4.070	3.870	4.300	3.720	4.450	4.080	4.250	103.455	0.000
mc1	7.933	**8.433**	4.650	3.850	3.767	4.517	3.783	4.033	4.033	107.854	0.000
mw1	6.100	**7.730**	4.670	4.770	4.830	4.220	3.530	4.680	4.470	48.230	0.000
pc1	8.430	**8.570**	3.380	4.270	4.350	3.320	3.920	4.600	4.170	132.182	0.000
pc2	2.750	3.380	4.620	**6.530**	6.050	6.080	5.650	5.080	4.850	51.709	0.000
pc3	8.100	**8.900**	3.650	4.250	3.880	4.550	3.730	4.120	3.820	129.866	0.000
pc4	8.080	**8.920**	3.580	3.880	4.170	4.270	2.970	4.630	4.500	135.484	0.000
pc5	8.200	**8.800**	3.870	3.730	3.630	4.230	3.830	4.570	4.130	129.298	0.000
ant-1.7	6.630	**7.530**	4.270	4.970	4.700	4.420	3.720	4.320	4.450	49.926	0.000
camel-1.6	8.200	**8.767**	4.080	3.630	4.850	4.250	3.220	3.800	4.200	132.108	0.000
ivy-2.0	**7.930**	3.430	5.520	4.900	4.050	4.580	4.970	4.600	5.020	50.490	0.000
jedit-4.3	**8.630**	8.130	3.850	3.850	3.850	3.850	5.130	3.850	3.850	213.087	0.000
log4j-1.2	1.067	2.533	5.700	5.850	5.783	6.250	5.550	**6.267**	6.000	112.974	0.000
lucene-2.4	**8.730**	8.230	3.800	3.920	3.930	4.120	4.420	4.230	3.620	127.274	0.000
poi-3.0	**8.700**	8.300	3.200	3.480	3.930	4.000	5.280	3.850	4.250	137.046	0.000
tomcat-6.0	7.700	**9.000**	3.630	3.650	3.880	4.520	3.900	4.500	4.220	122.274	0.000
xalan-2.6	8.400	**8.600**	3.520	4.700	3.880	4.020	3.500	4.600	3.780	131.768	0.000
xerces-1.4	8.370	**8.630**	3.030	3.580	3.970	3.470	6.430	3.850	3.670	155.979	0.000

8. Conclusions

In this paper, a hybrid classification approach for software defect prediction was proposed. The main idea of this approach was to develop expert and robust classification models based on groups of similar patterns. Two versions were developed and experimented on. The first was based on simple classifiers, whereas the second was based on ensemble ones. After extensive experiments based on 21 well-known benchmark datasets, the evaluation results showed that the ensemble version of the proposed approach can significantly boost the prediction power compared to the other ensemble and basic classifiers in most of the datasets. The reason for this superior performance is that the proposed approach develops models that fit specific patterns that have similar behaviours.

For future work, two areas could be researched for improvement. The first is to explore more advanced clustering algorithms, and the second is to investigate techniques that can automatically determine the best number of clusters for each dataset.

Author Contributions: Conceptualization, H.A. and H.F.; Methodology, H.A. and H.F.; Software, M.E. and N.H.; Validation, N.H., M.E. and I.A.; Formal analysis Y.A. and A.A.R.; Investigation, H.A. and N.H.; Resources, I.A. and A.A.R.; Data curation, H.A. and N.H.; Writing—original draft preparation, H.F. and H.A.; Writing—review and editing, I.A., H.A., Y.A. and H.F.; Supervision H.A. and H.F.; Project administration, H.F. and H.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rawat, M.S.; Dubey, S.K. Software defect prediction models for quality improvement: A literature study. *IJCSI Int. J. Comput. Sci. Issues* **2012**, *9*, 288–296.
2. Aljarah, I.; Banitaan, S.; Abufardeh, S.; Jin, W.; Salem, S. Selecting discriminating terms for bug assignment: A formal analysis. In Proceedings of the 7th International Conference on Predictive Models in Software Engineering, Banff, AB, Canada, 20–21 September 2011; p. 12.
3. Fenton, N.E.; Neil, M. Software metrics: Roadmap. In Proceedings of the Conference on the Future of Software Engineering, Limerick, Ireland, 4–11 June 2000; pp. 357–370.
4. Fenton, N.; Bieman, J. *Software Metrics: A Rigorous and Practical Approach*; CRC Press: Boca Raton, FL, USA, 2014.

5. Moser, R.; Pedrycz, W.; Succi, G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In Proceedings of the 2008 ACM/IEEE 30th International Conference on Software Engineering, Leipzig, Germany, 10–18 May 2008; pp. 181–190.
6. Bhattacharya, P.; Iliofotou, M.; Neamtiu, I.; Faloutsos, M. Graph-based analysis and prediction for software evolution. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 419–429.
7. Abaei, G.; Selamat, A. A survey on software fault detection based on different prediction approaches. *Vietnam J. Comput. Sci.* **2014**, *1*, 79–95. [CrossRef]
8. Wang, S.; Liu, T.; Tan, L. Automatically learning semantic features for defect prediction. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; pp. 297–308.
9. Clark, B.; Zubrow, D. *How Good Is the Software: A Review of Defect Prediction Techniques*; Sponsored by the US Department of Defense; InSoftware Engineering Symposium, Carreige Mellon University: Pittsburgh, PA, USA, 2001.
10. Hall, T.; Beecham, S.; Bowes, D.; Gray, D.; Counsell, S. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.* **2012**, *38*, 1276–1304. [CrossRef]
11. Malhotra, R. A systematic review of machine learning techniques for software fault prediction. *Appl. Soft Comput.* **2015**, *27*, 504–518. [CrossRef]
12. Menzies, T.; Milton, Z.; Turhan, B.; Cukic, B.; Jiang, Y.; Bener, A. Defect prediction from static code features: Current results, limitations, new approaches. *Autom. Softw. Eng.* **2010**, *17*, 375–407. [CrossRef]
13. Li, Z.; Reformat, M. A practical method for the software fault-prediction. In Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration, Las Vegas, IL, USA, 13–15 August 2007; pp. 659–666.
14. Vandecruys, O.; Martens, D.; Baesens, B.; Mues, C.; De Backer, M.; Haesen, R. Mining software repositories for comprehensible software fault prediction models. *J. Syst. Softw.* **2008**, *81*, 823–839. [CrossRef]
15. Mendes-Moreira, J.; Soares, C.; Jorge, A.M.; Sousa, J.F.D. Ensemble approaches for regression: A survey. *ACM Comput. Surv. (CSUR)* **2012**, *45*, 10. [CrossRef]
16. Alsawalqah, H.; Faris, H.; Aljarah, I.; Alnemer, L.; Alhindawi, N. Hybrid SMOTE-Ensemble Approach for Software Defect Prediction. In Proceedings of the Computer Science On-Line Conference, Prague, Czech Republic, 26–29 April 2017; Springer: Cham, Switzerland, 2017; pp. 355–366.
17. Rathore, S.S.; Kuamr, S. Comparative analysis of neural network and genetic programming for number of software faults prediction. In Proceedings of the 2015 National Conference on Recent Advances in Electronics & Computer Engineering (RAECE), Roorkee, India, 13–15 February 2015; pp. 328–332.
18. Rathore, S.S.; Kumar, S. Predicting number of faults in software system using genetic programming. *Procedia Comput. Sci.* **2015**, *62*, 303–311. [CrossRef]
19. Rathore, S.S.; Kumar, S. An empirical study of some software fault prediction techniques for the number of faults prediction. *Soft Comput.* **2016**, *21*, 7417–7434. [CrossRef]
20. Rathore, S.S.; Kumar, S. Linear and non-linear heterogeneous ensemble methods to predict the number of faults in software systems. *Knowl.-Based Syst.* **2017**, *119*, 232–256. [CrossRef]
21. Shatnawi, R.; Li, W. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *J. Syst. Softw.* **2008**, *81*, 1868–1882. [CrossRef]
22. Sandhu, P.S.; Singh, S.; Budhija, N. Prediction of level of severity of faults in software systems using density based clustering. In Proceedings of the 2011 IEEE International Conference on Software and Computer Applications. IPCSIT, Kathmandu, Nepal, 1–2 July 2011.
23. Menzies, T.; Greenwald, J.; Frank, A. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.* **2007**, *33*, 2–13. [CrossRef]
24. Suffian, M.D.M.; Ibrahim, S. A Prediction Model for System Testing Defects using Regression Analysis. *arXiv* **2014**, arXiv:1401.5830.
25. Koprinska, I.; Poon, J.; Clark, J.; Chan, J. Learning to classify e-mail. *Inf. Sci.* **2007**, *177*, 2167–2187. [CrossRef]
26. Elish, K.O.; Elish, M.O. Predicting defect-prone software modules using support vector machines. *J. Syst. Softw.* **2008**, *81*, 649–660. [CrossRef]

27. Huda, S.; Liu, K.; Abdelrazek, M.; Ibrahim, A.; Alyahya, S.; Al-Dossari, H.; Ahmad, S. An ensemble oversampling model for class imbalance problem in software defect prediction. *IEEE Access* **2018**, *6*, 24184–24195. [CrossRef]
28. Jiang, Y.; Cukic, B.; Ma, Y. Techniques for evaluating fault prediction models. *Empir. Softw. Eng.* **2008**, *13*, 561–595. [CrossRef]
29. Sun, Z.; Song, Q.; Zhu, X. Using coding-based ensemble learning to improve software defect prediction. *IEEE Trans. Syst. Man Cybern. Part C (Appl. Rev.)* **2012**, *42*, 1806–1817. [CrossRef]
30. El Emam, K.; Benlarbi, S.; Goel, N.; Rai, S.N. Comparing case-based reasoning classifiers for predicting high risk software components. *J. Syst. Softw.* **2001**, *55*, 301–320. [CrossRef]
31. Seliya, N.; Khoshgoftaar, T.M. Software quality estimation with limited fault data: A semi-supervised learning perspective. *Softw. Qual. J.* **2007**, *15*, 327–344. [CrossRef]
32. Catal, C.; Sevim, U.; Diri, B. Software fault prediction of unlabeled program modules. In Proceedings of the World Congress on Engineering, London, UK, 1–3 July 2009; Volume 1, pp. 1–3.
33. Yuan, X.; Khoshgoftaar, T.M.; Allen, E.B.; Ganesan, K. An application of fuzzy clustering to software quality prediction. In Proceedings of the 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology, Richardson, TX, USA, 24–25 March 2000; pp. 85–90.
34. Rathore, S.S.; Kumar, S. A study on software fault prediction techniques. *Artif. Intell. Rev.* **2019**, *51*, 255–327. [CrossRef]
35. Challagulla, V.U.; Bastani, F.B.; Yen, I.L. A unified framework for defect data analysis using the mbr technique. In Proceedings of the 2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06), Arlington, VA, USA, 13–15 November 2006; pp. 39–46.
36. Guo, L.; Cukic, B.; Singh, H. Predicting fault prone modules by the dempster-shafer belief networks. In Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Montreal, QC, Canada, 6–10 October 2003; pp. 249–252.
37. Catal, C. Software fault prediction: A literature review and current trends. *Expert Syst. Appl.* **2011**, *38*, 4626–4636. [CrossRef]
38. Zhou, T.; Sun, X.; Xia, X.; Li, B.; Chen, X. Improving defect prediction with deep forest. *Inf. Softw. Technol.* **2019**, *114*, 204–216. [CrossRef]
39. Quah, T.S.; Thwin, M.M. Application of neural networks for software quality prediction using object-oriented metrics. In Proceedings of the International Conference on Software Maintenance (ICSM 2003), Amsterdam, The Netherlands, 22–26 September 2003; pp. 116–125.
40. Evett, M.; Khoshgoftar, T.; Chien, P.D.; Allen, E. GP-based software quality prediction. In Proceedings of the Third Annual Conference Genetic Programming, Madison, WI, USA, 22–25 July 1998; pp. 60–65.
41. De Carvalho, A.B.; Pozo, A.; Vergilio, S.R. A symbolic fault-prediction model based on multiobjective particle swarm optimization. *J. Syst. Softw.* **2010**, *83*, 868–882. [CrossRef]
42. Koru, A.G.; Liu, H. Building effective defect-prediction models in practice. *IEEE Softw.* **2005**, *22*, 23–29. [CrossRef]
43. Qiu, S.; Lu, L.; Jiang, S.; Guo, Y. An investigation of imbalanced ensemble learning methods for cross-project defect prediction. *Int. J. Pattern Recognit. Artif. Intell.* **2019**, *33*, 1959037. [CrossRef]
44. Peng, Y.; Kou, G.; Wang, G.; Wu, W.; Shi, Y. Ensemble of software defect predictors: An AHP-based evaluation method. *Int. J. Inf. Technol. Decis. Mak.* **2011**, *10*, 187–206. [CrossRef]
45. Czibula, G.; Marian, Z.; Czibula, I. Software defect prediction using relational association rule mining. *Inf. Sci.* **2014**, *264*, 260–278. [CrossRef]
46. Catal, C.; Diri, B. Software defect prediction using artificial immune recognition system. In Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering, Innsbruck, Austria, 13–15 February 2007; pp. 285–290.
47. Catal, C.; Diri, B. A fault prediction model with limited fault data to improve test process. In *International Conference on Product Focused Software Process Improvement*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 244–257.
48. Kanmani, S.; Uthariaraj, V.R.; Sankaranarayanan, V.; Thambidurai, P. Object-oriented software fault prediction using neural networks. *Inf. Softw. Technol.* **2007**, *49*, 483–492. [CrossRef]
49. Begum, M.; Dohi, T. A Neuro-Based Software Fault Prediction with Box-Cox Power Transformation. *J. Softw. Eng. Appl.* **2017**, *10*, 288. [CrossRef]

50. Köksal, G.; Batmaz, İ.; Testik, M.C. A review of data mining applications for quality improvement in manufacturing industry. *Expert Syst. Appl.* **2011**, *38*, 13448–13467. [CrossRef]
51. Catal, C.; Sevim, U.; Diri, B. Practical development of an Eclipse-based software fault prediction tool using Naive Bayes algorithm. *Expert Syst. Appl.* **2011**, *38*, 2347–2353. [CrossRef]
52. Son, L.H.; Pritam, N.; Khari, M.; Kumar, R.; Phuong, P.T.M.; Thong, P.H. Empirical Study of Software Defect Prediction: A Systematic Mapping. *Symmetry* **2019**, *11*, 212. [CrossRef]
53. Challagulla, V.B.; Bastani, F.B.; Yen, I.L.; Paul, R.A. Empirical assessment of machine learning based software defect prediction techniques. *Int. J. Artif. Intell. Tools* **2008**, *17*, 389–400. [CrossRef]
54. Catal, C.; Diri, B. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Inf. Sci.* **2009**, *179*, 1040–1058. [CrossRef]
55. Kaur, M.J.; Pallavi, M. Data mining techniques for software defect prediction. *Int. J. Softw. Web Sci. (IJSWS)* **2013**, *3*, 54–57.
56. Kumar, M.A.; Gopal, M. Least squares twin support vector machines for pattern classification. *Expert Syst. Appl.* **2009**, *36*, 7535–7543. [CrossRef]
57. Agarwal, S.; Tomar, D. A feature selection based model for software defect prediction. *Assessment* **2014**, *65*. [CrossRef]
58. Tomar, D.; Agarwal, S. Prediction of defective software modules using class imbalance learning. *Appl. Comput. Intell. Soft Comput.* **2016**, *2016*, 6. [CrossRef]
59. Shukla, H.; Verma, D.K. A Review on Software Defect Prediction. *Int. J. Adv. Res. Comput. Eng. Technol. (IJARCET)* **2015**, *4*, 4387–4394.
60. Kumar Dwivedi, V.; Singh, M.K. Software Defect Prediction using Data Mining Classification Approach. *Int. J. Technol. Res. Appl.* **2016**, *4*, 31–35.
61. Huda, S.; Alyahya, S.; Ali, M.M.; Ahmad, S.; Abawajy, J.; Al-Dossari, H.; Yearwood, J. A Framework for Software Defect Prediction and Metric Selection. *IEEE Access* **2018**, *6*, 2844–2858. [CrossRef]
62. Bowes, D.; Hall, T.; Petrić, J. Software defect prediction: Do different classifiers find the same defects? *Softw. Qual. J.* **2018**, *26*, 525–552. [CrossRef]
63. Turhan, B.; Bener, A.B. Software Defect Prediction: Heuristics for Weighted Naïve Bayes. In Proceedings of the ICSOFT (SE), Barcelona, Spain, 22–25 July 2007; pp. 244–249.
64. Misirli, A.T.; Bener, A.B. A mapping study on bayesian networks for software quality prediction. In Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, Hyderabad, India, 3 June 2014; pp. 7–11.
65. Kotsiantis, S.B.; Zaharakis, I.; Pintelas, P. Supervised machine learning: A review of classification techniques. *Emerg. Artif. Intell. Appl. Comput. Eng.* **2007**, *160*, 3–24.
66. Freund, Y.; Schapire, R.E. Experiments with a new boosting algorithm. *Inicml* **1996**, *96*, 148–156.
67. Schapire, R.E. Explaining adaboost. In *Empirical Inference*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 37–52.
68. Freund, Y.; Schapire, R.E. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.* **1997**, *55*, 119–139. [CrossRef]
69. Breiman, L. Bagging predictors. *Mach. Learn.* **1996**, *24*, 123–140. [CrossRef]
70. Wang, T.; Li, W.; Shi, H.; Liu, Z. Software defect prediction based on classifiers ensemble. *J. Inf. Comput. Sci.* **2011**, *8*, 4241–4254.
71. Breiman, L. Random forests. *Mach. Learn.* **2001**, *45*, 5–32. [CrossRef]
72. Chen, T.; Guestrin, C. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd ACM Sigkdd International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 785–794.
73. Menzies, T.; Caglayan, B.; Kocaguneli, E.; Krall, J.; Peters, F.; Turhan, B. The Promise Repository of Empirical Software Engineering Data. Available online: http://promise.site.uottawa.ca/SERepositor (accessed on 1 April 2012).
74. Shepperd, M.; Song, Q.; Sun, Z.; Mair, C. Data quality: Some comments on the nasa software defect datasets. *IEEE Trans. Softw. Eng.* **2013**, *39*, 1208–1215. [CrossRef]

75. Ghotra, B.; McIntosh, S.; Hassan, A.E. Revisiting the impact of classification techniques on the performance of defect prediction models. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; Volume 1, pp. 789–800.
76. Lever, J.; Krzywinski, M.; Altman, N. Points of significance: Model selection and overfitting. *Nat. Methods* **2016**, 13, 703–704. [CrossRef]