Using Classifiers for Software Defect Detection

Logan Perreault, Seth Berardinelli, Clemente Izurieta, John Sheppard
Department of Computer Science, Montana State University
{logan.perreault,seth.berardinelli}@msu.montana.edu, {clemente.izurieta,john.sheppard}@montana.edu

***Abstract*—Classifier models have successfully been applied to the task of detecting and predicting defects in software projects. These models work by learning patterns between attributes of the software and a corresponding binary label that indicates whether or not a defect exists. In this study, we evaluate the performance of five different classifiers in the context of software defect detection – naïve Bayes, neural networks, support vector machines, logistic regression, and k-nearest neighbor. We measure performance using both accuracy and F_1-score, and use ANOVA to test for significance. Each classifier was run on five datasets from NASA's metrics data program repository. Results show that all models can detect software defects using static software features with a relatively high degree of certainty, although naive Bayes and support vector machines were outperformed by the remaining algorithms for some datasets.**

I. Introduction

Detection and prediction of software defects is an important problem in the software engineering community. Without an accurate method for detecting defects in software, a product may be released in an unsatisfactory condition. In addition, a method for predicting the occurrence of defects in software may actually prevent the degradation of software by suggesting corrective development that may avoid future problems.

One method for performing software defect detection is to construct a classification model of the software system. These models, which in the machine learning community are often just called "classifiers", typically consist of several attribute variables and a single class variable. These learned models capture the relationships between various software features and how they influence the class label, which in this case is a binary variable indicating that there is or is not a defect. Once learned, the model is able to predict whether or not a defect has occurred given a specific state instantiation to the features.

In addition to providing a means to detect and predict defects, models can help to better understand the underlying software features and how they affect the defect rate. For example, a model may indicate that features A and E have very little influence on software defects, while C may be extremely important. In addition, more complex relationships may arise where B and D may be important when considered together, but alone are poor indications of defects. This information can be extremely useful, and may suggest an area of focus for software design companies. Software companies are continuously improving and maturing existing code. If certain components are found to be defect-prone or likely to contain a fault in the near future, this may provide guidance as to where development efforts should be directed.

II. Related Work

Machine learning techniques have been applied to the software engineering domain in the past. One such work by Menzies *et al.* attempted to perform defect detection by making use of classifier models [1]. Specifically, the authors evaluate naïve Bayes (described in section III-A1, J48 (which is a decision tree model that builds prediction rules based on attribute values), and OneR (which can be viewed as a decision tree with only one level. Various modifications to the algorithms and preprocessing techniques were considered during the comparison. Performance was measured in terms of true positives and false negatives, which in this context equates to detect rates and false alarm rates respectively. The data chosen for experimentation was taken from NASA's metrics data program (MDP) repository. Results showed that in general, naïve Bayes tended to perform the best.

Challagulla *et al.* build on this work by analyzing a larger collection of machine learning techniques for the purpose of software defect detection [2]. The models studied by these authors are decision trees, naïve Bayesian networks, logistic regression, nearest neighbor, 1-rule, and neural networks. Here again, the data was taken from NASA's MDP repository. Results showed that naïve Bayes typically performed best, followed by neural networks.

Guo *et al.* use random forests to detect fault prone C and C++ modules [3]. These random forests consist of many decision trees constructed using subsets of the data. The datasets chosen were the same as those used in Challagulla *et al.*. The random forest was compared to 12 other classifiers taken from the WEKA, as well as to ROCKY, which is a defect detector toolset for NASA that consists of many singleton rules comparing attributes to a static threshold. Results showed that random forests generally outperformed other methods, and did not focus on comparison of the WEKA implementations against one another.

In similar work, Khoshgoftaar *et al.* use a Classification And Regression Trees (CART) algorithm to predict fault-prone modules [4]. The goal of their work is to reduce the number of misclassified fault-prone modules at the expense of a higher misclassification of modules that are not fault-prone. The logic is that a false alarm about a fault-prone module may be acceptable, but failing to detect a faulty model may have a more negative impact on a software company.

Finally, Zhou *et al.* use linear regression to detect varying degrees of software defects [5]. More specifically, the authors attempt to detect high-severity or low-severity faults, if such

a fault exists at all. This study again uses NASA datasets, but in this case the authors focus on object oriented design metrics. Interestingly enough, results show that the metrics that describe the software allow logistic regression to predict low-severity faults better than high-severity faults.

Similar to previous work, we compare naïve Bayesian networks, logistic regression, nearest neighbor and neural networks. We also use datasets from the NASA MDP repository – $PC1$, $PC2$, $PC3$, $PC4$ and $PC5$. We distinguish our work by including support vector machines, and by implementing our own versions of each of the algorithms. These unique implementations avoid the concern of biasing the WEKA implementations for each algorithm, which are the versions used by all previous studies [6]. Our work is therefore a pseudoreplication study, where we intend to verify some of the results presented by Challagulla *et al.*, but also modify the study to explore new classifiers as well.

III. Approach

The goal of this research is to determine whether or not the choice of classifier makes a difference when attempting to perform defect detection and prediction in software systems. To test this, we implement a variety of classifier models and compare their performance when applied to defect detection datasets. Throughout this study, we attempt to generally follow the framework proposed by Lessmann *et al.* [7]. In particular, we try to avoid implementation bias by writing our own algorithms, and use sufficient statistical testing procedures to verify our empirical findings.

A. Algorithms

For our experiments we implemented five different classifiers from machine learning and compared their performance in terms of their ability to detect software defects. These classifiers were chosen in an attempt to achieve diversity. The models make use of probabilities, connectionist methods, quadratic programming, regression and instance-based learning. In this way, we are able to explore a wide variety of possible classifier models using a representative subset. The implementations for each of these five algorithms are available on a publicly hosted on a GitHub repository [8]. A brief description for each algorithm is provided in this section.

1) Naïve Bayes: A Bayesian network is a probabilistic graphical model that compactly represents a joint probability distribution over a set of discrete random variables [9]. The model uses a directed acyclic graph where nodes represent variables and edges encode conditional independencies between these variables. Each node is said to be conditionally independent of all non-descendant nodes given its parents in the graph. An example of this is shown in Figure 1 [10].

Naïve Bayesian classifiers are a specialization of the model that work by imposing a specific network structure to the graph, where the class node is the sole parent to every feature node, and the class node has no parents of its own [11]. This corresponds to the assumption that every feature is conditionally independent of all other features given the class node. This structure allows the inference to be done very efficiently, since calculation of the class probability is simply a product of probabilities corresponding to each feature. If this assumption is not made, the number of values necessary to represent the probability distribution over binary variables becomes 2^{n+1}. This assumption gives the classifier the 'naïve' qualifier. While this would seem to be a strong assumption, empirical tests have shown that these models still perform very well in practice.

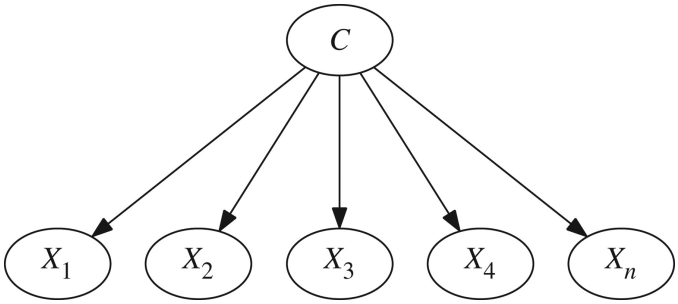


Fig. 1: A class node C with n attributes

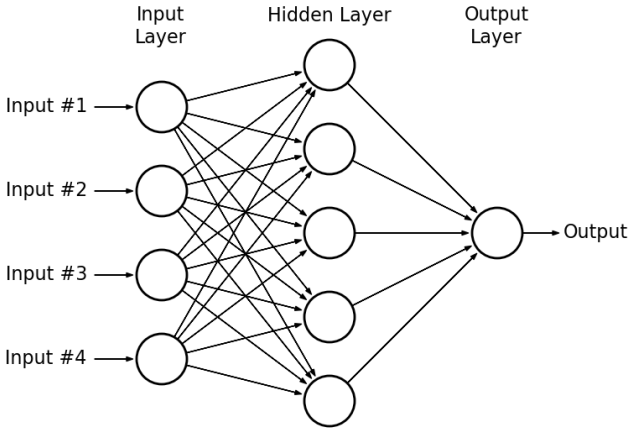


Fig. 2: An artificial neural network with a 4 node input layer, 5 node hidden layer, and 1 node output layer

2) Neural Network: Neural networks are models inspired by biological neural networks consisting of neurons and synapses. The model represents neurons as nodes in a graphical network, while the synapses become weighted edges between the nodes. Conceptually, the neurons act as processing units while the synapses pass information between neurons. It is this concept that inspires the mathematical model and the mechanics of the artificial neural network (ANN) classifiers, and perceptrons [12]. A fully-connected feed-forward ANN is comprised of layers, where the nodes in each layer are not connected, but the nodes between each layer are fully connected. An example of this type of graphical representation is shown in Figure 2.

These connections represent series of incoming weighted links that in the simplest case are considered to be "high" or

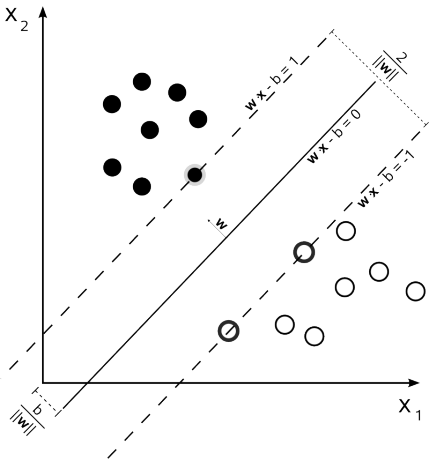


Fig. 3: A computed hyperplane (dark line) shown with a soft margin (area between dotted lines) separating the data points

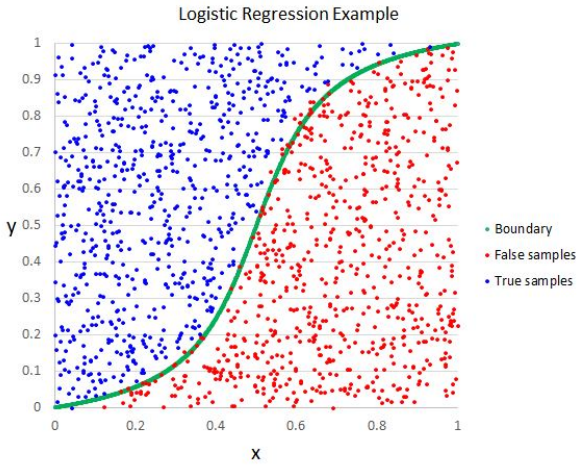


Fig. 4: A decision boundary found using Logistic Regression

"low" values produced by a step function. These incoming weighted values are summed and passed to the node's activation function as input. Typically, an ANN has an input layer, a hidden layer, and output layer, which in combination are capable of modeling complex functions. In practice, the step function is simulated using a differentiable sigmoid function, which allows the error gradient to be computed for learning. For classification, the last layer usually consists of binary class nodes, whose value is ultimately determined based on the inputs to the network. Backpropagation is the most common technique for learning edge weights, which works by adjusting values according to the gradient of error [13]. Effectively, this captures how the weights affect the "downstream" behavior of a node's input. Here, error is determined by incorrect classification produced in the output layer.

3) Support Vector Machine: A Support Vector Machine (SVM) aims to split data into two different spaces using a hyperplane [14]. When applied to classification, the goal is to find a hyperplane that provides a maximal margin between datapoints in different classes, as shown in Figure 3. The data points that *support* the position of the hyperplane are known as the support vectors. The problem of deciding these points is typically solving using Quadratic Programming.

SVM's are linear classifiers, meaning that it is unable to handle cases where class data overlaps. To account for this, Cortes and Vapnik introduced a Soft-Margin Hyperplane, which allows for class overlap by introducing a penalty for how far a particular data point breaches the hyperplane into the other class's feature space [15]. The penalty is captured more formally by introducing a slack variables to the Quadratic Programming formulation. Another way to handle non-linearly separable problems is to perform what is known as the "kernel trick", which lifts the data into a higher dimensional space by applying a transformation to the data [16]. Our research uses this method by applying a radial basis function as the kernel, which is the most popular choice of kernel for SVMs.

4) Logistic Regression: Another very popular classifier that deals with decision boundaries is Logistic Regression [17]. The main point of this model is to make decisions about the data. This is done by constructing a decision boundary (literally a curve or surface) among the data in order to, in our case, classify an example in one of two classes. Logistic Regressions is a probability model that captures the relationship between a set of independent variables and a dependent variable that represents the class. The coefficients of the logistic function need to be optimized in order to produce the decision boundary. Figure 4 shows an example of a decision boundary found using logistic regression [18].

Two popular methods exist for training a logistic regression model. The first is gradient descent (GD), which optimizes using a simple hill-climbing approach [19]. Second is the Newton-Raphson method (NR), which is more commonly used for multiclass regression but can still be used in the binary case [20]. This method is similar to gradient descent, but computes roots of the gradient function and follows a path to the root. In this work, we make use of the NR method.

5) K-Nearest Neighbor: K-Nearest Neighbor (k-NN) is a lazy learning method, meaning that there is no real learning phase of the algorithm [21]. A model is constructed by simply adding training examples. At any point during the addition of the training examples, the model is technically ready for testing examples. Classification occurs by first measuring the distance between the test point and all other points in the data set. Any distance metric can be used for this process, but Euclidean distance is a natural choice for many domains. The k nearest neighboring points to the test point are then selected, and the class is determined to be the one most common among this set of points.

A k-NN algorithm can be implemented a various ways. The most common data structure used for classification is the kd-tree [22]. A kd-tree works be dividing the space into a an d-dimensional tree. By traversing this tree, a point, x_i, can be said to be close to another point, x_j, if the point, x_i, exists

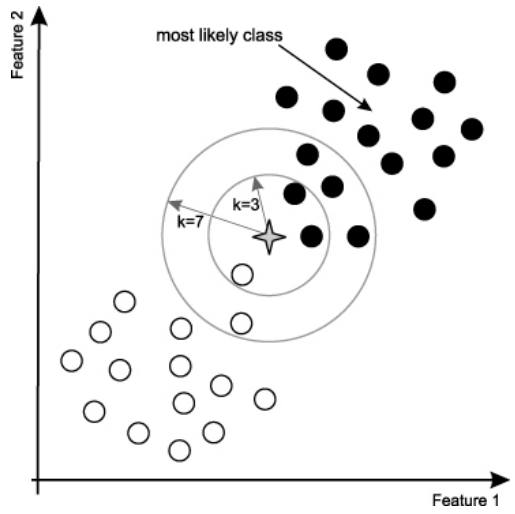


Fig. 5: Nearest Neighbor construction approximation using $k = 3$ and $k = 7$ for the data point represented by a star.

within divided space of the point, x_j. Another reason for such a representation is speed and efficiency. Traversing a kd-tree is often times much faster than brute force. For classification, a kd-tree becomes inefficient when the dimensionality of the feature-space becomes very large [1]. An example of how k-NN might work is shown in Figure 5 [23].

B. Data

The datasets used in this study were gathered from the tera-PROMISE Software Engineering Repository, which contains code for several [24]. This repository is an updated version of the PROMISE repository [25]. This study specifically focuses on datasets designed for defect detection, where each datapoint represents a snapshot of code. The attributes for each datapoint are intended to provide an indication of whether or not the code is defective. Attributes are represented with either discrete or continuous values, which may be simple measurements or more complex metrics. Several of the classifiers we use are not naturally suited for continuous variables, and we are therefore required to discretize the space. The achieve this, we performed equal-width binning with three bins for all attributes with a continuous state space.

One possible concern is that a classifier that makes use of poorly defined attributes will be difficult to interpret and reproduce in another setting. For this reason, the datasets chosen for this study are restricted to only those that have clearly defined metrics for attributes. Specifically, we look at datasets that make use of Halstead [26] and McCabe [27] metrics. Both McCabe and Halstead metrics consist of well defined measurements and calculations that are easily reproducible [28]. These specific metrics are a natural choice due to the fact that they have previously been used for software defect prediction [29].

[1]For this research, the dimensionality of feature-space becomes large when 20 or more attributes/features are used for classification.

TABLE I: Dataset Information

Dataset	Attributes	Pos. Examples	Neg. Examples	Total
PC1	38	61	644	705
PC2	37	16	729	745
PC3	38	134	943	1077
PC4	38	178	1280	1458
PC5	39	58	1826	1884

Five datasets were chosen from the tera-PROMISE repository, denoted throughout the remainder of this paper as $PC1$, $PC2$, $PC3$, $PC4$ and $PC5$. These datasets are also available at the NASA's MDP repository, although the corresponding versions are not identical [30]. $PC1$, $PC3$, and $PC4$ contain information about flight software written for an Earth orbiting satellite [31], [32]. $PC2$ describes a dynamic simulator for an altitude control system, and $PC5$ is for a safety enhancement to a cockpit upgrade system. Although similar, the datasets vary slightly in structure. All datasets are associated with code written in the C Programming Language, with the exception of $PC5$ which was written in C++. The attribute count and class statistics are shown in Table I. For more information regarding the specific attributes contained in each dataset, refer to the corresponding attribute-relation file format (ARFF) files in the tera-PROMISE repository.

IV. Experimental Design

There is some debate in the machine learning community about which metric best describes the classification capabilities of a model. Although a variety of metrics exist, the two most widely used include accuracy and the F_1 measure. Accuracy is the number of correctly classified instances divided by the total number of classified instances. Precision measures only the correctly classified positive examples divided by the total number of positive examples. Alternatively, recall measures the fraction of true positives divided by true positives and false negatives. The F_1 measure is defined as the harmonic mean of both precision and recall as shown in the equation below. The primary difference between accuracy and F_1 is that F_1 ensures that heavily imbalanced classes are not ignored.

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

To avoid biasing a single metric, we run two sets of experiments in parallel, each of which makes use of a different metric. For the accuracy experiments, the null hypothesis states that there is no difference between the choice of classifier with respect to classification accuracy. Similarly, for the F_1 experiments, the null hypothesis is that there is no difference between classifiers with respect to the F_1 measure. These hypotheses are designed to answer the question of which (if any) classifier is the correct choice for performing defect prediction in software systems. Note that the only difference between these hypotheses is the choice of response variable.

We ran experiments in a 10-fold cross validation scheme. Here, 90% of the data is used for training and the remaining

10% is used for testing. The data is then "rotated" so that a new 10% is chosen for testing. This is done 10 times (or folds) so that all data has been tested once. We repeat this procedure for every dataset, and do this for every classifier. All of this is repeated twice, once for each choice of response variable.

Most of the classifiers used in this study had a variety of hyperparameters that influenced its performance. The degree to which these parameters effect the resulting model's behavior varies for each classifier in that some are more sensitive to parameter changes than others. The parameters for each of these models was adjusted manually based on empirical evidence in an attempt to maximize performance for each model individually. Specifically, we reserved 25 examples from each dataset to use for manual testing when adjusting the algorithm parameters. The tuning process was accomplished by choosing a parameter and incrementally varying its value until performance is optimal for the 25 examples. This process is repeated for each of the parameters in turn, which assumes the parameters can be optimized individually. A more robust tuning process may consider all combinations of parameter values at once, but this is exponential in the number of possibilities. Cross validation was then performed on the remaining examples in the dataset. Once set, the parameters were kept constant throughout the varying folds and datasets.

V. Analysis

To get an initial idea of how each algorithm performs, we show the accuracy for each model on each of the datasets with 95% confidence bounds in Table II. Similar results are shown by Table III, except that the response variable is the F_1 measure instead of accuracy. These bounds were calculated based on standard error, and have been rounded to the nearest percent. In addition, we plot the accuracy and F_1 measure averaged over each of the folds and each of the datasets. This plot is shown in figure 6. The most immediately noticeable aspect of the data is the slight decrease in performance for the naïve Bayes model, especially for the accuracy response variable. The remaining algorithms appear fairly consistent. The difference between the using accuracy and F_1 as a response variable seems minimal in all cases. Although this information is interesting, no conclusions can be drawn that are not based on intuition. We investigate further by running significance tests. For our analysis, we follow the guidelines described by Juristo *et al.* [33]. Note that throughout this section, we abbreviate the classifiers as Naïve Bayes (NB), Neural Network (NN), Support Vector Machine (SVM), Logistic Regression (LR), and K-Nearest Neighbor (KNN).

In our experiments, we are dealing with a one-factor, k-alternative design. The factor in question depends on the experiment we are running. In the first set of experiments, we use accuracy as a response variable. The second set of experiments uses the F_1 metric. The k alternatives in this case refer to the six different classifiers. It is our belief that there are no significant blocking variables to consider. The linear model we use to describe this system is show below. Here, μ represents the grand mean of all variables, α_j is the mean

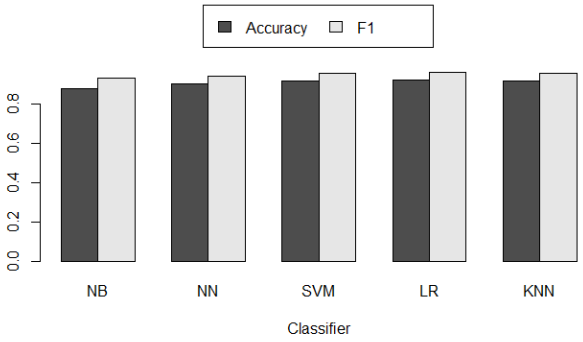


Fig. 6: Mean classifier performance over all datasets

for the j^{th} model, and e_{ij} is the error associated with the i^{th} fold and the j^{th} model. y_{ij} is either the accuracy or the F_1 measure of the model, depending on the experiment.

$$y_{ij} = \mu + \alpha_j + e_{ij} \tag{1}$$

To validate this model, we ensure that the data is normal independent and identically distributed (n.i.i.d.), which involves checking for both normality and independence. To perform the normality check, we ensure that the residuals generally follow a normal distribution. The residuals are calculated by subtracting each observation from its expected value, which is simply the mean for the alternative. The plot of the residuals for the $PC5$ dataset is shown in figure 7. We have performed this check for all datasets with similar results but omit all figures due to space constraints. To check for independence, the residuals are plotted as a function of the expected observation values. If there is no discernible trend in the graph, then the residuals can be assumed to be independent. The residual plot is omitted due to space constraints, however there was no observable trends in the charts, regardless of the dataset.

We performed an ANOVA test to compare the six alternatives. As normality and independence assumptions have been validated, we are justified in choosing an ANOVA test. A test was run for each of the five datasets, and for each of the response variables for a total of 10 tests. The results of these tests are summarized in table IV, which provides the resulting p-values. In every case, the difference in the response variable is significant, even at a 0.001 significance level.

One concern is that the ANOVA test does not explain which of the classifiers specifically makes the most difference. Instead we run a Tukey's test to determine which of the classifier models are causing significant differences. This test compares each combination of alternatives in a pairwise manor, which we do for each of the datasets. To display this information compactly, we include the dataset name in each cell where the alternatives differs at a 0.05 significance level. This information is displayed as upper triangular matrices in tables V and VI for accuracy and F_1 respectively.

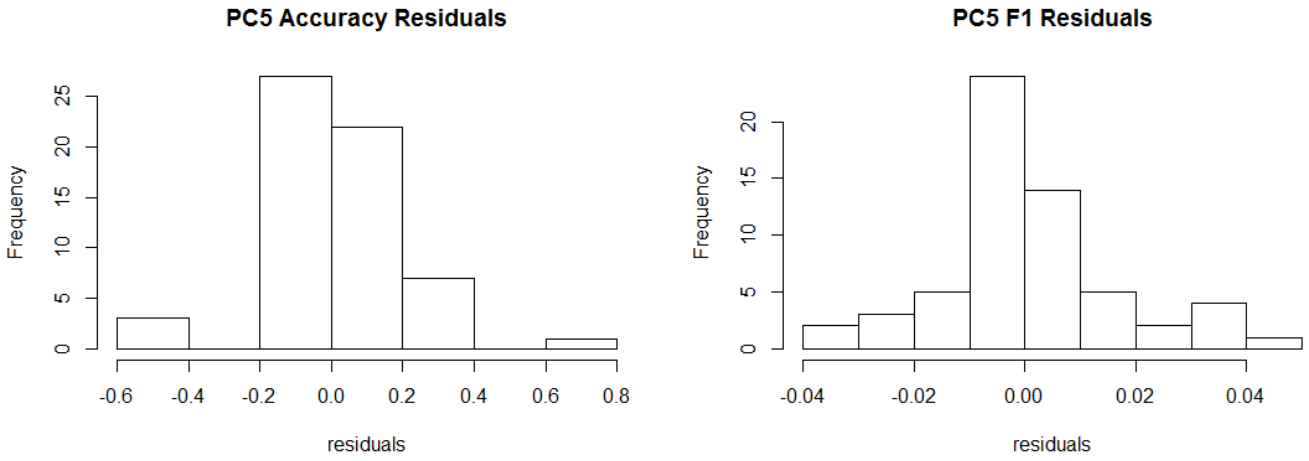
The results from the Tukey test are somewhat more informative than the ANOVA test alone. Specifically, we see that regardless of the response variable, naïve Bayes is significantly

TABLE II: Confidence Interval for Accuracy in percentages

	PC1	PC2	PC3	PC4	PC5
NB	0.84 ± 0.02	0.93 ± 0.01	0.83 ± 0.01	0.83 ± 0.01	0.96 ± 0.01
NN	0.91 ± 0.01	0.98 ± 0.01	0.88 ± 0.01	0.79 ± 0.09	0.95 ± 0.01
SVM	0.91 ± 0.01	0.98 ± 0.01	0.88 ± 0.02	0.87 ± 0.01	0.94 ± 0.01
LR	0.91 ± 0.01	0.98 ± 0.01	0.88 ± 0.02	0.88 ± 0.01	0.97 ± 0.00
KNN	0.91 ± 0.01	0.98 ± 0.01	0.86 ± 0.02	0.87 ± 0.01	0.97 ± 0.00

TABLE III: Confidence Interval for F_1 in percentages

	PC1	PC2	PC3	PC4	PC5
NB	0.91 ± 0.01	0.96 ± 0.04	0.91 ± 0.01	0.90 ± 0.01	0.98 ± 0.00
NN	0.95 ± 0.00	0.99 ± 0.00	0.93 ± 0.01	0.84 ± 0.09	0.98 ± 0.01
SVM	0.95 ± 0.01	0.99 ± 0.00	0.93 ± 0.01	0.93 ± 0.00	0.97 ± 0.01
LR	0.95 ± 0.01	0.99 ± 0.00	0.93 ± 0.01	0.93 ± 0.00	0.98 ± 0.00
KNN	0.95 ± 0.00	0.99 ± 0.00	0.92 ± 0.01	0.93 ± 0.00	0.98 ± 0.00



(a) Accuracy metric

(b) F_1 metric

Fig. 7: Normality check for the $PC5$ dataset

TABLE IV: p-values for the ANOVA tests

	PC1	PC2	PC3	PC4	PC5
Accuracy	< 0.000	< 0.000	0.138	0.004	0.039
F_1	< 0.000	< 0.000	0.086	< 0.000	0.035

TABLE V: Tukey significant accuracy differences

	NN	SVM	LR	KNN
NB	PC1, PC2, PC4	PC1, PC2, PC4	PC1, PC2, PC4	PC1, PC2
NN		0	0	0
SVM			PC5	PC5
LR				0
KNN				

TABLE VI: Tukey significant F_1 differences

	NN	SVM	LR	KNN
NB	PC1, PC2, PC4	PC1, PC2, PC4	PC1, PC2, PC4	PC1, PC2, PC4
NN		0	0	0
SVM			PC5	PC5
LR				0
KNN				

different from all other classifiers for $PC1$ and $PC2$. This is also true for the $PC4$ dataset, except in the case of k-nearest neighbor when using accuracy as a response variable. Next we see that SVMs are significantly different from logistic regression and k-nearest neighbor classifiers on the $PC5$ dataset with respect to both accuracy and the F_1 measure. Neural networks, logistic regression, and k-nearest neighbor do not show significant differences between one another.

VI. Discussion

The results from the ANOVA tests verify that the choice of classifier does in fact influence both response variables at a significance level of 0.05 for datasets $PC1$, $PC2$, $PC4$, and $PC5$. This indicates that for these datasets, the change observed in the response variables due to the choice of classifier is very unlikely to occur by chance. This means

we can safely reject both of the null hypotheses listed above. The $PC3$ dataset does not show any differences between the algorithms unless we consider a 0.1 significance level, and even then is only significant when using the F_1 measure as a response variable. The implication is that some care needs to be dedicated to researching and choosing a proper classification model in order to improve the ability to predict defects in software.

In addition, the Tukey results show that using naïve Bayes generally has a significant effect on the overall classification performance, and the same is true for SVMs on the $PC5$ dataset. By looking at the performances from tables II and III, we see that in the cases where these classifiers differ, their performance is lower than the competitors. This somewhat contradicts the results presented by Challagulla *et al.*, which indicates that naïve Bayes typically performs better [2].

Since the performance degradation seen by naïve Bayes and SVM is known to be significant, the obvious choice is to avoid using these classifiers for the task of defect detection for related projects. This general recommendation is based on the notion that naïve Bayes and SVM performed worse in some situations, and never performed better. Interestingly enough however, we did not observe a classifier that performs worse over all five datasets. Given that these datasets describe similar projects all written in the C Programming Language, it may seem surprising that classifiers do not behave similarly between projects. This difference suggests that a classifier must be chosen on a project-by-project basis. It also may indicate that a mixture of classifiers that considers the results from a set of classifiers may be more effective than a single classifier on its own.

A. Threats to Validity

One threat to the validity of this study is the choice of metric used for the response variable. Although widely used in the machine learning community, accuracy and F_1 measures may not be the best choice for representing the effectiveness of software defect classifiers. Some other commonly used alternatives include an F_2 measure, which weights recall higher than precision, and $F_{0.5}$, which does the exact opposite. Another popular choice is to measure the area under the receiver operating characteristic (ROC) curve, which plots the true positive rate against the false positive rate. Finally, another option may be to use precision or recall as a measure on its own. We believe reasonable attempts have been made to avoid this construction threat, but acknowledge that additional response variable metrics would improve our results.

Another potential problem is that all experiments were run on similar datasets. Although there are differences between the datasets that appear to have an impact on classifier performance, all projects are also very similar in that they are NASA software projects written in C or C++. To gain a better understanding of how classifiers behave in the general case, it would be best to diversify the type of datasets that are used. One possible option is to look at software projects written in another language like Java. Without doing so, the results are not guaranteed to apply to scenarios outside of the studied domain. With that said, a correct choice for a classifier model can only be made in the context of a specific domain. This stems from the "no free lunch" theorem in machine learning, that states that any two classifiers are equivalent when considered over all domains [34]. Our results support this hypothesis, in that the differences between classifiers changed depending on the considered dataset.

We personally implemented each of the six classifiers that were compared in this study. Although we have tested these algorithms on other problems with good results, there is the potential that there exists bugs within our own software, or that the implementations are sub-optimal in some way. In addition, it may make reproducing our results somewhat difficult. Other works have made use of WEKA, which is a commonly used implementation of many classifiers that makes comparison between works very simple. This may explain the discrepancy between our work and Challagulla *et al.* regarding naïve Bayes. Although results obtained using WEKA are easily reproducible, it may lead to research directions biased toward the WEKA implementations of an algorithm. For this reason, we believe it is important to study a variety of algorithmic implementation so as not to optimize for one standard implementation. In addition, we have stored our code in a public repository to address the concern of reproducibility when using personal implementations [8].

VII. Conclusion

We have implemented and tested five different classifiers from the field of machine learning: naïve Bayes, neural networks, support vector machines, logistic regression, k-nearest neighbor. We applied these models to the task of defect prediction, using NASA's $PC1$, $PC2$, $PC3$, $PC4$ and $PC5$ datasets. Results show that there is in fact a difference in the choice of classifier with respect to the model's ability to classify. More specifically, it is very likely that different classifiers will produce different accuracy and F_1 scores. Moreover, we determined that our implementation of naïve Bayes was outperformed by the remaining algorithms in three of the datasets, while SVM was outperformed by logistic regression and k-nearest neighbor for one of the datasets.

These results are primarily exploratory. More research is needed to make a truly informed decision on which classifier is best suited for performing defect detection in software. We have made recommendations on how to choose a classifier for defect detection, and have argued for the infeasibility of choosing a generally optimal classifier for all software projects. Future work will analyze the performance of a mixture of classifiers, which should smooth some of the differences in classifier performance. We also hope to increase the number of analyzed classifiers, as well as the number and variety of datasets on which they are tested. In this way, we hope to uncover an underlying structure of software projects that consistently produces significant differences in the classifiers. Although no single classifier can be chosen that performs

best in all scenarios, specific software project types may be identified in which there is a correct choice of classifier.

Another potential avenue for future research is the continued study of the work done by Challagulla *et al*. It is somewhat surprising that our results do not coincide with their work, but many aspects of the experiment were changed. We used personal implementations rather than WEKA, and used different datasets as well (although there was some overlap). To account for these changes, a separate study could be conducted in the future that attempts to replicate Challagulla's work as closely as possible, and compare results in that fashion.

Our results support previous work in this area, showing that classifiers can indeed be used to effectively predict defects using static software features. The classifiers can be used to identify defect-prone components in a software project, and given the proper choice in classifier can do so with a relatively high degree of certainty. The information we obtain from these classifiers can be extremely effective for the continuous improvement of software, in that it may direct code maturation efforts. Alternatively, classifiers may help to avoid running unit tests for an entire software project, which is often quite costly with respect to time. Instead, a nightly test may only consider components that are predicted to be faulty, and omit the rest.

Although it still seems to be an open question as to which classifier is the optimal choice, it is certainly clear that the choice of classifier does affect performance. Effective defect prediction models are important for determining and preventing problems with software. Our research shows that when attempting to perform defect detection, attention must be paid to which classifier is to be used. The choice of model is a difficult problem that requires more research. Until this area of research is more thoroughly developed, choosing defect prediction models will be forced to rely on domain-specific empirical testing to make the proper choice.

REFERENCES

[1] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007.

[2] V. U. B. Challagulla, F. B. Bastani, I.-L. Yen, and R. A. Paul, "Empirical assessment of machine learning based software defect prediction techniques," *International Journal on Artificial Intelligence Tools*, vol. 17, no. 02, pp. 389–400, 2008.

[3] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *15th International Symposium on Software Reliability Engineering. ISSRE 2004.* IEEE, 2004, pp. 417–428.

[4] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Classification tree models of software quality over multiple releases," in *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*. IEEE, 1999, pp. 116–125.

[5] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *Software Engineering, IEEE Transactions on*, vol. 32, no. 10, pp. 771–789, 2006.

[6] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[7] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 485–496, 2008.

[8] S. Berardinelli and L. Perreault, "Software bug detection," https://github.com/sdberardinelli/sw-bug-detection, 2015.

[9] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[10] D. T. Pham and G. A. Ruz, "Unsupervised training of bayesian networks for data clustering," in *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 465, no. 2109. The Royal Society, 2009, pp. 2927–2948.

[11] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification (2Nd Edition)*. Wiley-Interscience, 2001.

[12] M. Minsky and S. Papert, *Perceptron: an introduction to computational geometry (expanded edition)*. The MIT Press, Cambridge, 1969.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, 1988.

[14] M. Andriansyah, A. Suhendra, and I. Wicaksana, "Comparative study: The implementation of machine learning method for sentiment analysis in social media. a recommendation for future research," *Advanced Science Letters*, vol. 20, no. 10-12, pp. 2009–2013, 2014.

[15] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[16] A. Aizerman, E. M. Braverman, and L. Rozoner, "Theoretical foundations of the potential function method in pattern recognition learning," *Automation and remote control*, vol. 25, pp. 821–837, 1964.

[17] D. R. Cox, "The regression analysis of binary sequences," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 215–242, 1958.

[18] D. Snider. (2015) Introduction to the sql server analysis services logistic regression data mining algorithm. [Online]. Available: http://www.mssqltips.com/sqlservertip/3471/introduction-to-the-sql-server-analysis-services/logistic-regression-data-mining-algorithm/

[19] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2009.

[20] D. Böhning, "Multinomial logistic regression algorithm," *Annals of the Institute of Statistical Mathematics*, vol. 44, no. 1, pp. 197–200, 1992.

[21] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[22] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[23] D. L. A. AL-Nabi and S. S. Ahmed, "Survey on classification algorithms for data mining:(comparison and evaluation)," *Computer Engineering and Intelligent Systems*, vol. 4, no. 8, pp. 18–24, 2013.

[24] O. Baysal, A. Benar, T. Menzies, and B. Turhan, "The tera-PROMISE Repository of Software Engineering Databases - Version 4." NC State University, 2014. [Online]. Available: http://openscience.us/repo/

[25] 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

[26] M. H. Halstead, "Natural laws controlling algorithm structure?" *ACM Sigplan Notices*, vol. 7, no. 2, pp. 19–26, 1972.

[27] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[28] B. Curtis, S. B. Sheppard, P. Milliman, M. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics," *IEEE Transactions on Software Engineering*, no. 2, pp. 96–104, 1979.

[29] T. M. Khoshgoftaar and J. C. Munson, "Predicting software development errors using software complexity metrics," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 2, pp. 253–261, 1990.

[30] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the nasa software defect datasets," *Software Engineering, IEEE Transactions on*, vol. 39, no. 9, pp. 1208–1215, 2013.

[31] T. Menzies and J. S. Di Stefano, "How good is your blind spot sampling policy," in *Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004.* IEEE, 2004, pp. 129–138.

[32] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow, "Comparing design and code metrics for software quality prediction," in *Proceedings of the 4th international workshop on Predictor models in software engineering*. ACM, 2008, pp. 11–18.

[33] N. Juristo and A. M. Moreno, *Basics of software engineering experimentation*. Springer Publishing Company, Incorporated, 2010.

[34] D. Wolpert, "The lack of a priori distinctions between learning algorithms," *Neural Computation*, vol. 8, no. 7, pp. 1341–1390, 1996.