Predicting Android Application Risk Levels Using Code Metrics

Akond Rahman, Arjun Datt Sharma, Alexander Rouse, Andrew Buckley, Ezra Zerihun and Priysha Pradhan
Department of Computer Science, North Carolina State University,
Raleigh, North Carolina, USA
{aarahman, asharm24, aprouse, ajbuckle, eczerihu, ppradha3}@ncsu.edu

***Abstract* – Whether or not an Android application is benign or malicious, the application can be susceptible to security and end-user privacy related risks. Automated techniques that identify Android applications or components within Android applications can help Android application developers to create less risky applications. *The goal of this project is to aid Android application developers in assessing the risk associated with developed Android applications by using code metrics as predictors to predict different levels of risk for Android applications*. In our study we investigate if code metrics such as code complexity, lines of code per application, and number of classes per application, can be used to predict various levels of risk for Android applications. We use simple logistic regression model to identify necessary code metrics that can predict risk levels of Android applications. We use five classification techniques namely Decision Tree, Gaussian Naïve Bayes classifier, Nearest Neighbor classifiers, Random Forests, and Support Vector Machines to predict different levels of risk using the identified code metrics. Furthermore, we study the effects of different parameters on performance prediction for the five classification techniques. From our study we identify 21 code metrics that can be used to predict risk levels of Android applications. We observe that with respect to precision and recall, Random Forests is a better classification technique. We also observe that with respect to mean classification error and average accuracy, Random Forests is significantly better than the four other classification techniques. Furthermore, we observe that the parameters of the classification techniques have an impact on prediction performance. Findings from our work suggests that similar to defect prediction for software modules, code metrics can be used effectively to predict risk levels for Android applications with proper selection of classification technique, as well as, appropriate tuning of parameters for the classification technique of interest.**

Git Link: https://github.ncsu.edu/asharm24/Android-Risk-Classification.git

TABLE OF CONTENT

I. INTRODUCTION

Wide spread popularity of Android applications comes with the price of security and privacy risk for the end-users. Regardless of being classified as malware or benign, Android applications have the potential of posing security and privacy risks for end-users [1]. Android developers unwillingly might create Android applications that re-use third party libraries containing vulnerabilities, contain infected binaries, or use permission settings that expose private health and monetary information of the end-user [1]. In order to reduce security and privacy risk of developed Android applications, security professionals have advocated of creating futuristic tools that will facilitate in determining the amount of security and privacy risk that is associated with the developed applications [2]. Prior studies have showed the effectiveness of using code metrics in predicting failure prone components, in a traditional software development setting [3] [4]. But the Android application development process is different to that of the traditional software development process [5], and whether or not code metrics can be useful for predicting risk levels of Android applications requires systematic exploration. In our work we investigate if code metrics that are available from static analysis tools, can be used to predict various levels of risk associated with Android applications.

The goal of this project is to aid Android application developers in assessing the risk associated with developed Android applications by using code metrics as predictors to predict different levels of risk for Android applications.

In our work we focus our effort in answering the following research questions to achieve this goal:

RQ-1: What code metrics can be used as predictors to predict different levels of risk for Android applications?

RQ-2: What machine learning techniques can be used to predict different levels of risk using code metrics?

RQ-3: How do the parameters of machine learning techniques used to predict different levels of risk effect prediction performance?

II. BACKGROUND AND RELATED WORK

We organize this section in two categories: first we briefly describe the contents of the dataset that we use in this project. Next we provide an overview of the two major types of content namely risk scores, and code metrics that we included in this project from the dataset of interest. Finally we explore and briefly present prior academic work that is closely related to our project.

A. Dataset

In our work we rely on prior work to resolve this necessity and use a dataset published by Krutz et al. [6]. According to Krutz et al. [6], the goal of the publishing this dataset was to facilitate the security specific research in the domain of Android applications. The dataset included code metrics such as number of lines of code, functional complexity, and cyclomatic complexity, as well as metrics related to the Git history of each of the Android applications of interest. Furthermore, Krutz et al. collected risk scores of the Android applications. Krutz et al. used static analysis tools such as Stowaway and SonarQube to collect code and Git metrics, and used Androrisk to collect the risk scores of various versions of Android applications. In total the dataset included 4416 versions of 1179 open source Android applications. For each of these 4416 versions of Android applications the dataset includes risk scores, and values for 21 code metrics. The authors organized the dataset based on versions of the applications i.e. the code metrics, risk scores, and Git history related metrics are available for the versions of applications instead of the Android application themselves. In our project we used two features of the published dataset: the code metrics and the risk scores. In the following segments we describe what code metrics we included in the study with necessary definitions. We also briefly describe the concept of risk scores for Android applications, and how they are quantified.

B. Contents of the Dataset Used in the Project

As previously stated we used two major types of contents from the study namely, risk scores, and code metrics. In this subsection first we present an overview of the risk score, then we briefly describe the code metrics used in the study.

B.1. Code Metrics Used in the Project

The goal of our study is to investigate how code metrics such as functional complexity, number of lines of code per application, and number of classes per application can be used to predict different levels of risk. We discuss our approach in Section III in details, but present the definitions of the 21 code metrics considered in the project below:

Number of classes: This code metric corresponds to the number of classes in the Android application [8].

Lines of code: This metric represents Number of physical lines that contain at least one character excluding white spaces, tabs, or strings that are part of a comment [8].

Methods: This metric represents the number of Java methods in total for the application [8].

Duplicated blocks: A portion of the segment is considered a duplicated block if at least 10 successive and duplicated statements regardless of the number of tokens and lines. Duplicated blocks correspond to the count of duplicated blocks of lines [8].

Duplicated Lines: This metric corresponds to the count of physical lines involved in duplication [8].
Duplicated Files: This metric corresponds to the count of files involved in duplication [8].
Complexity: This metric is also known as McCabe's complexity or cyclomatic complexity that describes how many times the control flow of function is splitted across the application [9].
Class Complexity: This metric refers to the average amount of McCabe's complexity for each class in the Android application [8].
Function Complexity: This metric refers to the average amount of McCabe's complexity for each method in the Android application [8].
Comment lines: This metric refers to the count of lines that includes one or multiple comments excluding comments that only include special symbols such as @, #, and $ [8].
Density of Comment Lines: This metric is measures as (Comment Lines / (Lines of code + Comment lines)) * 100
Lines: This metric means the number of physical lines including carriage returns in the application [8].
Percentage of comments: This metric is computed as following: (comment lines / (lines of code + comment lines) * 100) [8].

Percentage of duplicated lines: This metric is computed as following: (Duplicated lines / Lines * 100) [8].
Files: This metric corresponds to the number of files in the application [8].
Directories: This metric corresponds to the number of directories or folders in the application.
File Complexity: This metric refers to the average amount of McCabe's complexity for each file in the Android application [8].

Violations: SonarQube quantifies the count of issues existing in the application that might hinder productivity for the team as well as for the developer, and potentially induce security flaws. This metric refers to the count of such issues quantified by SonarQube. SonarQube considers five types of violations namely, Blocker, Critical, Major, Minor, and Info [8]. The dataset does not include the metric information for Info, and therefore we present the definitions of the other four metrics as following.
Blocker Violations: This metric corresponds to the number of 'blocker' violations within the application. SonarQube categorizes a violation as blocker if the violation corresponds to an issue that might make the whole application unstable in production such as, not closing a socket, and calling the garbage collector object [8].
Critical Violations: This metric corresponds to the number of 'critical' violations within the application. SonarQube categorizes a violation as critical if the violation corresponds to an issue that might lead to unexpected behavior e.g. a Null Pointer Exception, without jeopardizing the integrity of the whole application [8].
Major Violations: This metric corresponds to the number of 'major' violations within the application. SonarQube categorizes a violation as major if the violation corresponds to an issue that might lead to lower productivity. In SonarQube, reduced productivity refers to tasks that require extra effort that is not directly related to the task itself for example, working with complex methods, working through packing cycles etc. [8].
Minor Violations: This metric corresponds to the number of 'minor' violations within the application. SonarQube categorizes a violation as minor if the violation corresponds to an issue that does not lead to lower productivity [8].

C. Related work

Our project is closely related to prior academic studies that have investigated on detection and prediction of Android malware, as well as have investigated the end-user privacy issues associated with Android applications.

Existing work had heavily focused on detecting and categorizing malicious Android applications using Android permission files [10]–[12], dynamic analysis [13], [14], and applying statistical and machine learning techniques [15], [16]. Yang et al. [13] introduced DroidMiner that tracks Framework API calls executed by Android applications and used that information to detect and classify malicious Android applications. Pandita et al. [17] investigated to what extent descriptions of Android applications match with that of the requested permissions using WHYPER, a tool that uses applies natural language techniques. Peng et al. [15] stated the in-effectiveness of permission-based alerting to inform end-users about the risks of Android applications and proposed multiple variants of Naïve Bayes models that categorizes market place Android applications as being highly or less risky. Suarez-Tangil et al. [16] adopted a different approach and used text-mining techniques to categorize and synthesize the evolution of android malware families.

Our project takes a different approach compared to these prior research methods. In our work, we use code metrics as predictors predict different levels of security and privacy risks associated with open source Android applications.

III. RESEARCH METHODOLOGY

In this section we describe the steps to conduct our study. The major steps of this study are presented in Figure 1. The major steps of the study include pre-processing of the dataset, creating and assigning labels for each Android application version within the pre-processed dataset using clustering techniques, selecting appropriate code metrics as predictors, predicting levels of risk for the Android versions with multiple prediction models and finally investigating the effects of different combination of parameters for each classification model used in the project. We describe each of these processes in the following subsections.

D. Dataset Processing:

From our preliminary exploration, we observe that the original dataset in its raw format is susceptible to include application versions that have no risk scores, and include application versions that are small in size e.g. an application version can have only one source code file. Based on these observations we apply the following selection criteria to create the formatted dataset that will be used to conduct the rest of the study:

1. The application version must have a non-zero risk score i.e. any application version that has a zero risk score will not be included in the formatted dataset.
2. The application version must contain at least *f* number of files, where *f* is the median of the number of files for each application version, belonging to the original dataset.

By applying the above-mentioned criteria we collect code metrics, risk scores for the application versions. We refer to this pre-processed dataset as the *formatted dataset* for the rest of this project.

E. Categorizing Multiple Levels of Risk Using Clustering Techniques:

In our study we focus on predicting different levels of risk using code metrics instead of predicting the risk scores. We state that multiple levels of risk associated with each application version such as 'high', 'medium', and 'low', might convey more information than the original risk score for the application version. However, determining the levels of risk scores is non-trivial as Krutz et al. [6] did not provide any indication on what scores can be considered as 'high', 'medium', or 'low'. We apply hierarchical clustering technique to categorize the levels of risk for application versions. As we have no prior knowledge about the categorization of risk scores, and therefore about the quality of the clusters, we use three cluster validation techniques to determine the appropriate labels. These three cluster validation measures are connectivity, Dunn index, and Silhouette width. The goal of using these three measures is to find the optimal number of clusters that can be used to categorize the risk scores. For example, according to the Connectivity measure is the optimal number of clusters are three, then we can consider three levels of risk namely 'high', 'medium', and 'low'.

We observe the possibility of obtaining different conclusions by using these three different cluster validation measures. For example, according to Silhouette Width we might find that five is the optimal number of clusters whereas, whereas, according to Connectivity, two is the optimal number of clusters. In such cases we consider all conclusions i.e. we separately label the formatted dataset using two labels and five labels. We apply our feature selection scheme and prediction scheme on both of these differently labeled datasets: the dataset labeled using two levels, and the dataset labeled using five levels.

In our study we use the R-package 'clvalid' [29] to compute the cluster evaluation techniques. In our study we observe how the three cluster evaluation measures vary for $N/5$ clusters, where N is the total number of records in the formatted dataset. From our analysis of cluster evaluation techniques we obtain the number of clusters we need to label the data for prediction. Next, we use the Scikit Learn API [32] to apply agglomerative clustering to create the n number of clusters on the formatted dataset, where n is the number of optimal clusters determined by the analysis of cluster evaluation. From the output of hierarchical clustering we obtain labels for the n clusters that are translated into different levels. The labels are numeric values that correspond to the average of the risk scores within a cluster. We create labels by translating these numeric values into different categories. This labeled dataset will be used for feature selection and predicting the levels of risk. Empirical findings related to this step are available in Section IV.

F. Selecting Appropriate Code Metrics Using Feature Selection

Our formatted dataset with appropriate labels contains 21 code metrics that we want to use for predicting the class labels. We used Logistic regression to find best attributes. In the weighted equation of regression, each of the predictors used in training is assigned a weight using an *l1* or *l2* penalty scheme. Predictors in the equation that have zero weights are considered as weak predictors, whereas predictors with non-zero weights are considered to be strong predictors of the predicted value.

We apply *l2* penalized logistic regression model on the formatted dataset with labels to categorize the predictors that we will use for prediction. In our project we consider those code metrics as predictors that have non-zero weights in the fitted equation generated by the logistic regression model. We use the Scikit Learn API [32] to implement the *l2* penalized logistic regression model.

G. Predicting Levels of Risk Using Prediction Models:

Using the obtained predictors in the previous step we build prediction models to predict different levels of risk for application versions. Please recall that we use clustering techniques to determine the levels of application version risk. Considering this fact, we frame the prediction of risk levels as a multi-class classification problem, where each class corresponds to a level of the risk. Regardless of how many levels of risk scores are categorized from our prior analysis, our procedure for predicting the levels of risk remains the same.

We select five prediction models that belong to five different categories. These prediction models are Classification and Regression Trees (CART) that belong to the 'Decision tree' category, Radial-Basis Support Vector Machine (R-SVM) that belong

to the ‘Support vector machine-based’ category, Random Forests (RF) that belong to ‘Ensemble’ approaches, k Nearest Neighbor (kNN) that belong to the ‘Nearest neighbor’ approach, and Gaussian Naïve Bayes predictor (GNB) that belongs to the ‘Naïve-based’ category [37].

We evaluate the performance of the five prediction models by applying k-fold cross validation technique. K-fold cross validation is a technique to test the performance of predictors where k-1 of the k folds are used as training data, and the remaining fold is used as test data [43]. According to prior work [43] we observe the selection of k in k-fold cross validation impacts prediction accuracy. Therefore, we choose two different values for k-fold cross validation to determine if the difference in training size has an impact on prediction performance. We apply two and five-folded cross validation for all five predictors, and all levels of risk. We use the Scikit Learn API [32] to implement cross validation techniques, the five prediction models, and the prediction performance measures that are discussed below.

For each of the prediction models we use three metrics to determine the prediction accuracy. These three metrics as precision, recall, and F-measure. Furthermore, to determine if the performance of any of the five prediction models is significantly worse than the other four, we use mean absolute error (MAE) [45] that measures the weighted average of all mis-predicted levels of risk.

H. Investigating the Effects of Parameters on Prediction Performance

Most prediction models and mining algorithms have configuration parameters that can be tuned to obtain desirable results. As we use the Scikit Learn API to implement our prediction models, we are dependent on the parameters that are available in Scikit Learn. Also, depending on the prediction model. Some parameters might not be applicable to our project. Considering these facts we select a specific set of parameters for each of the five prediction models that are explained in detail in the Appendix section.

For all the five prediction models we run prediction experiments using all possible combinations of the selected parameter values. Similar to our approach described in the previous Section III-D, we measure prediction performance for each of these experiments using precision, recall, F-measure, and MAE.

IV. EMPIRICAL FINDINGS

We use this section to present our findings that answer our research questions of interest. We also present empirical findings that we obtained by executing each step of our methodology.

I. Data Processing

Previously in Section III-A we have described how we create the formatted dataset by applying two criteria. Using our selection criteria, we created the formatted dataset that had 549 versions of application with risk scores, and 21 code metrics.

J. Categorization of Risk Scores

We have applied hierarchical clustering to categorize the levels of risk from the risk score residing in the formatted dataset. We have used three measures namely, connectivity, Dunn Index, and Silhouette Width to determine the quality of the clustering. We observe that the quality of clustering varies with the amount of clusters considering the three techniques namely Connectivity, Dunn Index, and Silhouette Width, as shown in Figures 2, 3, and 4 respectively. We observe that the best clustering quality using Connectivity, Dunn Index, and Silhouette Width respectively is obtained for two, 13, and 12 clusters. According to our methodology, we consider all of these clusters as levels of risk, and use these levels for prediction using code metrics. We also provide the amount of application versions that belong to each of these levels in Table I, indicated in the ‘Count’ column. In Table I, for 12 and 13 clusters, the levels with the lowest index correspond to the lowest level of the risk, whereas the highest index corresponds to the highest level of risk. For example, according to Table I, the lowest possible level of risk is ‘L0’, considering 12 clusters, and six application versions have a risk of level L0.

Figure 2: Connectivity Scores Increase with Number of Clusters. Connectivity is the lowest for Two Clusters.

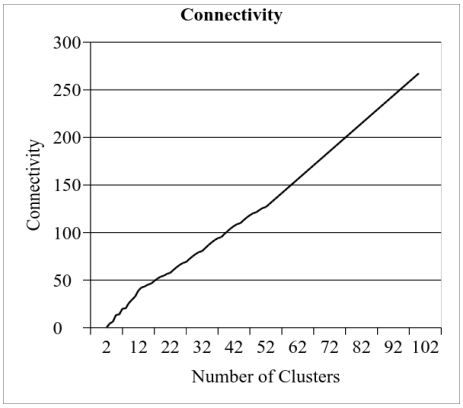


Figure 3: Dunn Index if the Highest For 13 Clusters

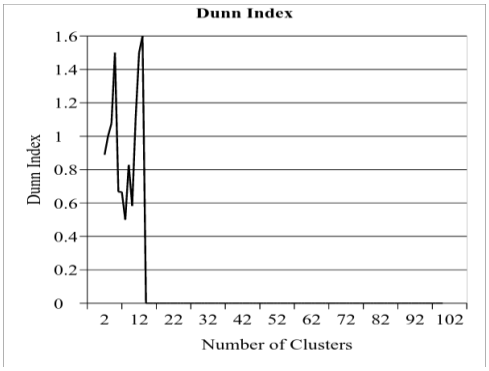
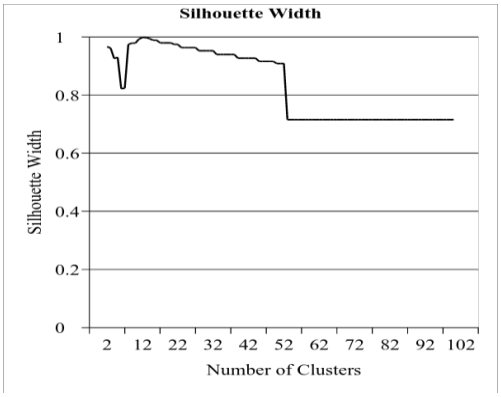


Figure 4: Silhouette Width is the Highest for 12 Clusters



Levels	Label of the Levels	Count
Two	High	532
	Low	17
12	L11	6
	L10	3
	L9	2
	L8	6
	L7	393
	L6	7
	L5	106
	L4	6
	L3	3
	L2	7
	L1	4
	L0	6
13	L12	6
	L11	3
	L10	2
	L9	5
	L8	1
	L7	393
	L6	7
	L5	106
	L4	6
	L3	3
	L2	7
	L1	4
	L0	6

Table 1: Levels of Risk of Application Versions

K. Feature Selection

Previously we have mentioned that how we have used logistic regression to identify the predictors that can be used to classify the levels of risk of application versions. Empirical findings from our application of logistic regression will also provide answers to RQ-1. We have observed that according to three different cluster evaluation techniques the levels of risk for application versions are two, 12, and 13. According to our methodology we will apply logistic regression for these three levels, and observe if the coefficient for predictors are non-zero. We will not include the predictors that have zero coefficients to train the classification model of interest.

Answer to RQ-1:

We do not observe any non-zero coefficients for any of the 21 predictors considering two, 12, and 13 levels of risks for application versions. As a representative example, in Table 2 we present the coefficient scores for all the 21 predictors.

Overall, we answer RQ-1 by making the following observation: the 21 code metrics have non-zero effects and all of them can be used as predictors to predict various levels of risk scores.

L. Classification models

CART Classification
CART classification was performed on the dataset with 2, 12 and 13 classes and 2 and 5-Fold cross validation. The results included the values of Precision, Recall and F-Measure obtained with both default as well as tuned parameters.
Table 3 in Appendix shows the results of CART for both default and tuned parameters. The best results for 2-Fold cross-validation were obtained for following parameter combinations:
For 2 Class : criterion=gini splitter=random max_features=5 max_depth=100 min_samples_split=20 min_samples_leaf=2 max_leaf_nodes=10000
For 12 Class : criterion=entropy splitter=random max_features=1 max_depth=100 min_samples_split=4 min_samples_leaf=4 max_leaf_nodes=500
For 13 Class : criterion=gini splitter=random max_features=1 max_depth=10 min_samples_split=2 min_samples_leaf=1 max_leaf_nodes=500

The best results for 5-Fold cross-validation were obtained for following parameter combinations:
For 2 Class : criterion=entropy splitter=best max_features=5 max_depth=10000 min_samples_split=8 min_samples_leaf=2 max_leaf_nodes=200
For 12 Class : criterion=entropy splitter=random max_features=1 max_depth=1000 min_samples_split=2 min_samples_leaf=1 max_leaf_nodes=100
For 13 Class: criterion=gini splitter=random max_features=5 max_depth=1000 min_samples_split=4 min_samples_leaf=1 max_leaf_nodes=500

Random Forests Classification
Random Forests classification was performed on the dataset with 2, 12 and 13 classes and 2 and 5-Fold cross validation. The results included the values of Precision, Recall and F-Measure obtained with both default as well as tuned parameters.

Table 4 in Appendix shows the results of Random Forests for both default and tuned parameters. The best results for 2-Fold cross validation were obtained for following parameter combinations:
For 2 Class : n_estimators=10, criterion=gini, max_features=sqrt, max_dept=None, max_leaf_nodes=75, bootstrap=False, min-sample-split=1, oob_score=False, min-wt-frac=0.2, warm-start=False
For 12 Class : n_estimators=50, criterion=entropy, max_features=sqrt, max_dept=15, max_leaf_nodes=75, bootstrap=True, min-sample-split=1, oob_score=False, min-wt-frac=0.0, warm-start=False
For 13 Class : n_estimators=100, criterion=gini, max_features=sqrt, max_dept=15, max_leaf_nodes=50, bootstrap=True, min-sample-split=1, oob_score=False, min-wt-frac=0.0, warm-start=False

The best results for 5-Fold cross validation were obtained for following parameter combinations:
For 2 Class : n_estimators=10, criterion=gini, max_features=auto, max_dept=15, max_leaf_nodes=None, bootstrap=False, min-sample-split=1, oob_score=False, min-wt-frac=0.2, warm-start=False
For 12 Class : n_estimators=500, criterion=entropy, max_features=sqrt, max_dept=None, max_leaf_nodes=75, bootstrap=True, min-sample-split=1, oob_score=False, min-wt-frac=0.0, warm-start=False
For 13 Class : n_estimators=100, criterion=entropy, max_features=auto, max_dept=30, max_leaf_nodes=None, bootstrap=True, min-sample-split=1, oob_score=False, min-wt-frac=0.0, warm-start=True

Support Vector Machine Classification
Support Vector Machine classification was performed on the dataset with 2, 12 and 13 classes and 2 and 5-Fold cross validation. The results included the values of Precision, Recall and F-Measure obtained with both default as well as tuned parameters.
Table 5 in Appendix shows the results of SVM (Support Vector Machine) for both default and tuned parameters. SVM did not show any improvement in the values of Precision, Recall, F-Measure or Mean Absolute Error after tuning the parameters.

Figure 5 and 6 show the comparison between average values of Precision, Recall and F-measure of all classification models for 2, 12 and 13 classes. Detailed results with values for each class can be seen from the tables 3,4,5,6 and 7 in appendix.

Gaussian Naive Bayes Classification

Gaussian Naive Bayes classification was performed on the dataset with 2, 12 and 13 classes and 2 and 5-Fold cross validation. The results included the values of Precision, Recall and F-Measure. No parameter tuning is available for GNB in scikit-learn library. Table 7 in Appendix shows the detailed results of Navie Bayes with default parameters..

K-Nearest Neighbors Classification

KNN classification was performed on the dataset with 2, 12 and 13 classes and 2 and 5-Fold cross validation. The results included the values of Precision, Recall and F-Measure obtained with both default as well as tuned parameters.

Table 6 in Appendix shows the results of KNN (K-Nearest Neighbors) for both default and tuned parameters. The best results for 2-Fold cross-validation were obtained for following parameter combinations:

For 2 Class : n_neighbors=10, weights=uniform, metric=euclidean, p=3, algorithm=auto
For 12 Class : n_neighbors=50, weights=distance, metric=euclidean, p=3, algorithm=auto
For 13 Class : n_neighbors=50, weights=distance, metric=euclidean, p=3, algorithm=auto

The best results for 5-Fold cross-validation were obtained for following parameter combinations:

For 2 Class : n_neighbors=50, weights=distance, metric=euclidean, p=3, algorithm=auto
For 12 Class : n_neighbors=50, weights=distance, metric=manhattan, p=3, algorithm=auto
For 13 Class: n_neighbors=50, weights=distance, metric=manhattan, p=3, algorithm=auto

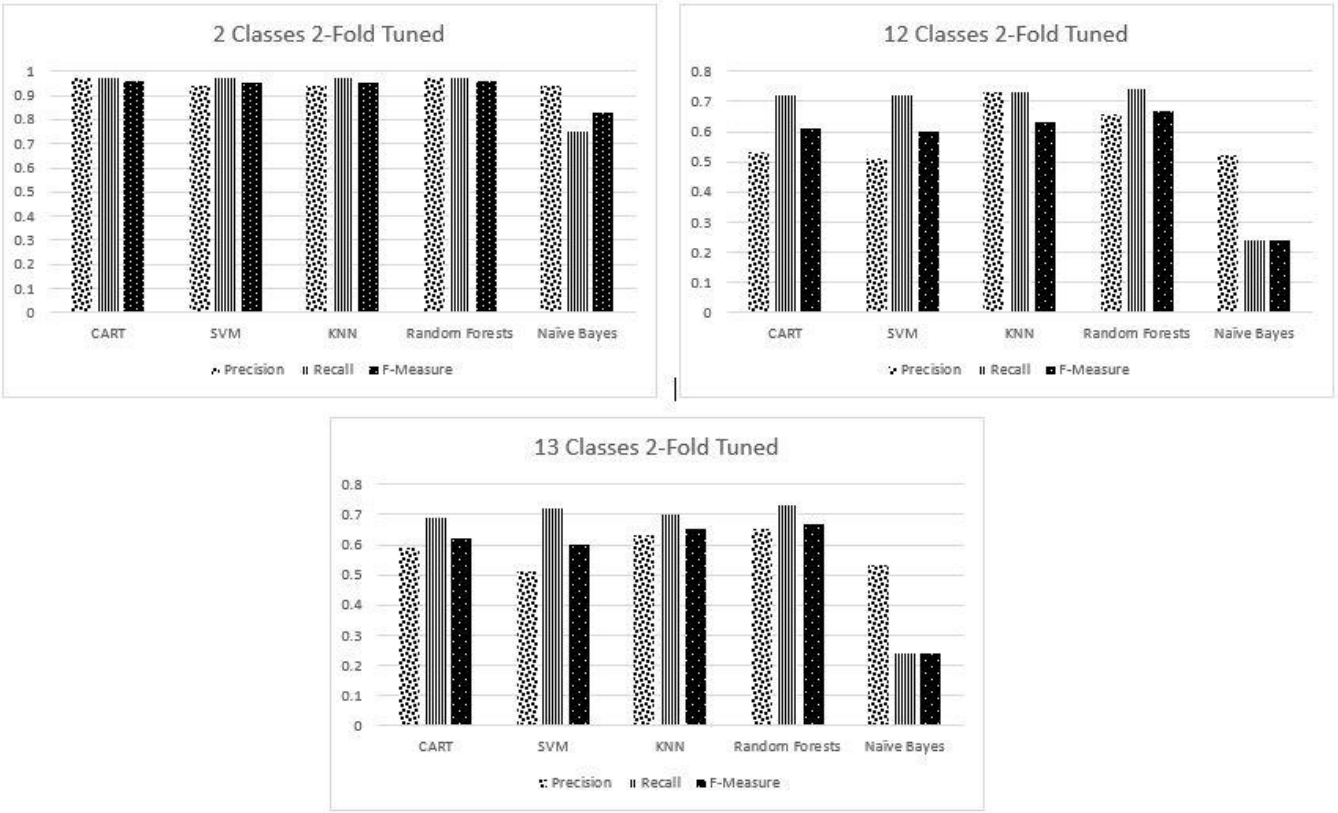


Figure 5: Comparison between average values of Precision, Recall and F-Measure of CART, SVM, KNN, Random Forests, Naive Bayes for 2-Fold

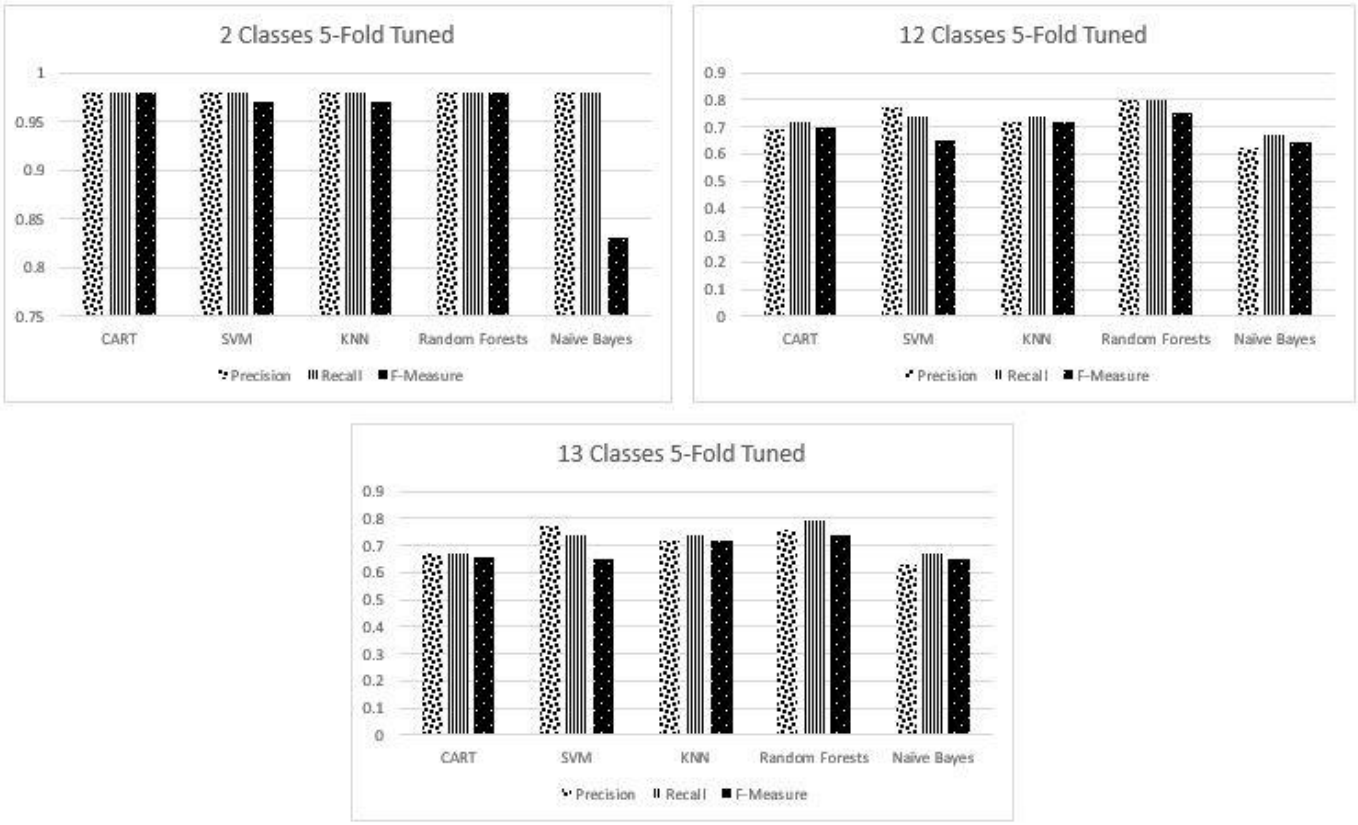


Figure 6: Comparison between average values of Precision, Recall and F-Measure of CART, SVM, KNN, Random Forests, Naive Bayes for 5-Fold

V. LIMITATIONS

We used two criteria to identify the 19 adoptees that we used in our study. These two criteria may generate false positives and false negatives.

VI. CONCLUSION AND FUTURE WORK

During the cluster validation experiment, results showed that Connectivity gave best result for 2 clusters, Silhouette Width for 12 clusters and Dunn Index for 13 clusters.

From the results of our classification experiments, it can clearly be concluded that Random Forests performed the best and gave the best scores for precision, recall, f-measure and mean absolute error. This conclusion was made by comparing values for each class given in Tables 3,4,5,6 and 7. For certain classification models, tuning of parameters improved the results. Following are the parameters that improved the performance of classification models:

CART : criterion ; Random Forests : number of estimates, impurity measure ; KNN : number of nearest neighbors.

On analyzing the table 5 in appendix, we concluded that SVM did not show any improvement in values of precision, recall, f-measure and mean absolute error on tuning the parameters. No tuning of parameters could be performed for Naive Bayes as scikit-learn did not have any parameters for GNB library.

Future work consists of collecting more data and performing oversampling and under-sampling on our dataset for more accurate results.

REFERENCES

[1] A. Atzeni, T. Su, M. Baltatu, R. D'Alessandro, and G. Pessiva, "How Dangerous is Your Android App?: An Evaluation Methodology," in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, ICST, Brussels, Belgium, Belgium, 2014, pp. 130–139.

[2] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, "Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2015, pp. 248–259.

[3] N. Nagappan, T. Ball, and A. Zeller, “Mining Metrics to Predict Component Failures,” in *Proceedings of the 28th International Conference on Software Engineering*, New York, NY, USA, 2006, pp. 452–461.

[4] N. Nagappan and T. Ball, “Static Analysis Tools As Early Indicators of Pre-release Defect Density,” in *Proceedings of the 27th International Conference on Software Engineering*, New York, NY, USA, 2005, pp. 580–586.

[5] A. I. Wasserman, “Software Engineering Issues for Mobile Application Development,” in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, New York, NY, USA, 2010, pp. 397–400.

[6] D. E. Krutz, M. Mirakhorli, S. A. Malachowsky, A. Ruiz, J. Peterson, A. Filipski, and J. Smith, “A Dataset of Open-source Android Applications,” in *Proceedings of the 12th Working Conference on Mining Software Repositories*, Piscataway, NJ, USA, 2015, pp. 522–525.

[7] K. Dunham, S. Hartman, J. Morales, M. Quintans, and T. Strazzere, *Android Malware and Analysis*. Boca Raton,FL: Taylor and Francis, 2014.

[8] G. A. Campbell and P. P. Papapetrou, *SonarQube in Action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2013.

[9] T. J. McCabe, “A Complexity Measure,” *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[10] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, “Checking App Behavior Against App Descriptions,” in *Proceedings of the 36th International Conference on Software Engineering*, New York, NY, USA, 2014, pp. 1025–1035.

[11] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, “A Methodology for Empirical Analysis of Permission-based Security Models and Its Application to Android,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2010, pp. 73–84.

[12] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, “Exploring Permission-Induced Risk in Android Applications for Malicious Application Detection,” *Trans Info Sec*, vol. 9, no. 11, pp. 1869–1882, Nov. 2014.

[13] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, “Computer Security - ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I,” M. Kuty\lowski and J. Vaidya, Eds. Cham: Springer International Publishing, 2014, pp. 163–182.

[14] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, “RiskRanker: Scalable and Accurate Zero-day Android Malware Detection,” in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, New York, NY, USA, 2012, pp. 281–294.

[15] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, “Using Probabilistic Generative Models for Ranking Risks of Android Apps,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, New York, NY, USA, 2012, pp. 241–252.

[16] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, “Dendroid: A Text Mining Approach to Analyzing and Classifying Code Structures in Android Malware Families,” *Expert Syst Appl*, vol. 41, no. 4, pp. 1104–1117, Mar. 2014.

[17] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, “WHYPER: Towards Automating Risk Assessment of Mobile Applications,” in *Proceedings of the 22Nd USENIX Conference on Security*, Berkeley, CA, USA, 2013, pp. 527–542.

[18] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, “Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications,” in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[19] V. Moonsamy, J. Rong, and S. Liu, “Mining permission patterns for contrasting clean and malicious android applications,” *Future Gener. Comput. Syst.*, vol. 36, pp. 122 – 132, 2014.

[20] V. Babu Rajesh, P. Reddy, P. Himanshu, and M. U. Patil, “DROIDSWAN: DETECTING MALICIOUS ANDROID APPLICATIONS BASED ON STATIC FEATURE ANALYSIS,” *Comput. Sci. Inf. Technol.*, p. 163.

[21] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, “Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets.,” in *NDSS*, 2012.

[22] P. H. Chia, Y. Yamamoto, and N. Asokan, “Is This App Safe?: A Large Scale Study on Application Permissions and Risk Signals,” in *Proceedings of the 21st International Conference on World Wide Web*, New York, NY, USA, 2012, pp. 311–320.

[23] A. Mylonas, M. Theoharidou, and D. Gritzalis, “Risk Assessment and Risk-Driven Testing: First International Workshop, RISK 2013, Held in Conjunction with ICTSS 2013, Istanbul, Turkey, November 12, 2013. Revised Selected Papers,” T. Bauer, J. Gro\s smann, F. Seehusen, K. Stølen, and M.-F. Wendland, Eds. Cham: Springer International Publishing, 2014, pp. 21–37.

[24] Y. Jing, G. J. Ahn, Z. Zhao, and H. Hu, “Towards Automated Risk Assessment and Mitigation of Mobile Applications,” *IEEE Trans. Dependable Secure Comput.*, vol. 12, no. 5, pp. 571–584, Sep. 2015.

[25] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, “Android Permissions: A Perspective Combining Risks and Benefits,” in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, New York, NY, USA, 2012, pp. 13–22.

[26] P.-N. Tan, M. Steinbach, V. Kumar, and others, *Introduction to data mining*, Indian Subcontinent Edition., vol. 1. UP, India: Pearson Addison Wesley Boston, 2006.

[27] P. Ferragina and A. Gulli, “A personalized search engine based on Web-snippet hierarchical clustering,” *Softw. Pract. Exp.*, vol. 38, no. 2, pp. 189–225, 2008.

[28] P. Treeratpituk and J. Callan, “Automatically Labeling Hierarchical Clusters,” in *Proceedings of the 2006 International Conference on Digital Government Research*, San Diego, California, USA, 2006, pp. 167–176.

[29] G. Brock, V. Pihur, S. Datta, and S. Datta, “clValid: An R Package for Cluster Validation,” *J. Stat. Softw.*, vol. 25, no. 1, pp. 1–22, 2008.

[30] L. J. Deborah, R. Baskaran, and A. Kannan, “A survey on internal validity measure for cluster validation,” *IJCSES*, vol. 1, no. 2.

[31] Y. Liu, Z. Li, H. Xiong, X. Gao, and J. Wu, “Understanding of Internal Clustering Validation Measures,” in *2010 IEEE International Conference on Data Mining*, 2010, pp. 911–916.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine Learning in Python,” *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[33] I. Guyon and A. Elisseeff, “An Introduction to Variable and Feature Selection,” *J Mach Learn Res*, vol. 3, pp. 1157–1182, Mar. 2003.

[34] M. Choetkiertikul, H. K. Dam, T. Tran, and A. Ghose, “Characterization and Prediction of Issue-related Risks in Software Projects,” in *Proceedings of the 12th Working Conference on Mining Software Repositories*, Piscataway, NJ, USA, 2015, pp. 280–291.

[35] S. Lee, H. Lee, P. Abbeel, and A. Y. Ng, “Efficient l1 regularized logistic regression,” in *In AAAI-06*, 2006.

[36] T. Menzies, J. Greenwald, and A. Frank, “Data Mining Static Code Attributes to Learn Defect Predictors,” *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[37] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, “Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings,” *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul. 2008.

[38] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[39] G. H. John and P. Langley, “Estimating Continuous Distributions in Bayesian Classifiers,” in *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, San Francisco, CA, USA, 1995, pp. 338–345.

[40] P. Cunningham and S. J. Delany, “k-Nearest neighbour classifiers,” *Mult. Classif. Syst.*, pp. 1–17, 2007.

[41] B. Schölkopf, K.-K. Sung, C. J. Burges, F. Girosi, P. Niyogi, T. Poggio, and V. Vapnik, “Comparing support vector machines with Gaussian kernels to radial basis function classifiers,” *Signal Process. IEEE Trans. On*, vol. 45, no. 11, pp. 2758–2765, 1997.

[42] L. Breiman, “Random forests,” *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[43] R. Kohavi and others, “A study of cross-validation and bootstrap for accuracy estimation and model selection,” in *Ijcai*, 1995, vol. 14, pp. 1137–1145.
[44] D. Powers, “Evaluation: From Precision, Recall and F Factor to ROC, Informedness, Markedness & Correaltion,” *Sch. Inform. Eng. Flinders Univ. S. Aust. Adel.*, 2007.
[45] C. J. Willmott and K. Matsuura, “Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance,” *Clim. Res.*, vol. 30, no. 1, p. 79, 2005.
[46] A. Vargha and H. D. Delaney, “A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong,” *J. Educ. Behav. Stat.*, vol. 25, no. 2, pp. 101–132, 2000.
[47] A. Arcuri and L. Briand, “A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering,” in *Proceedings of the 33rd International Conference on Software Engineering*, New York, NY, USA, 2011, pp. 1–10.
[48] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, “Static test case prioritization using topic models,” *Empir. Softw. Eng.*, vol. 19, no. 1, pp. 182–212, 2012.

APPENDIX

Existing work: Risk Score of Android Applications

Similar to Krutz et al. [6], and Atzeni et al. [1], in our paper we differentiate between a malware and focus on the amount of risk associated with each Android application. An Android malware is an Android application that exposes sensitive end-user information. An application can be benign or a malware, but still has some level of risk associated with it as the application can use special permissions [reff]. In our study, we use a dataset that includes risk scores of open source Android applications obtained from the Androrisk utility [reff to book]. Androrisk is a reverse engineering tool that takes the Android application’s Dalvik Executable (DEX), and APK files as input, and quantifies how risky the application by tracking the application’s use of shared libraries, third-party code bases, and requested permissions [7].

Parameters for different classification models:

CART: the parameters that we used are:

- criterion: the criterion technique to select a split. The Scikit Learn API allows two String values: ‘gini’, and ‘entropy’. We use both of these values.
- splitter: the strategy to select the split for each node. The Scikit Learn API allows two String values: ‘best’, and ‘random’. We use both of these values.
- max_features: presents the number of features to consider when making a split. The Scikit Learn API allows multiple values. We use ‘auto’, ‘sqrt’, ‘log2’, and None.
- max_depth: The maximum allowed depth of the decision tree. The Scikit Learn API allows multiple integer values. We use
- min_samples_split: Presents the minimum number of samples required to split an internal node. The Scikit Learn API allows integer values. We use
- min_samples_leaf : Presents the minimum number of samples required to be at the leaf nodes of the decision tree. The Scikit Learn API allows multiple integer values. We use
- max_leaf_nodes: Presents how many leaves will be used per node when creating the decision tree in a best first fashion. The Scikit Learn API allows multiple integer values. We use

kNN: The parameters that we used are:

- n_neighbors: Presents the number of neighbors that will be used to create the kNN prediction model. The Scikit Learn API allows multiple integer values. We use 1, 10, 50, and 100.
- weights: Presents the weight function that will be used to create the kNN prediction model. The Scikit Learn API allows two String values ‘uniform’, and ‘distance’ and one callable object. We use ‘uniform’, and ‘distance’.
- algorithm: Presents the algorithm that will be used to create the kNN prediction model. The Scikit Learn API allows four String values ‘auto’, ‘ball_tree’, ‘kd_tree’, and ‘brute’. We use all the four String values.
- metric: Presents the type of metrics that will be used to create the kNN prediction model. The Scikit Learn API allows four String values and ‘DistanceMetric’ type object. We use the four strings ‘euclidean’, ‘manhattan’, ‘chebyshev’, and ‘minkowski’.
- p: the power parameter when ‘minkowski’ is provided for the ‘metric’ parameter. The Scikit Learn API allows multiple integer values. We use 3, 5, and 10.

GNB: The Scikit Learn API does not allow any parameters for creating the GNB model and therefore we do not tune any parameters for GNB.

R-SVM: the parameters that we used are:

- C: Presents the penalty that will be used in creating the decision boundaries for r-SVM. The Scikit Learn API allows multiple integer values. We use 1, 10, 50, and 100.

- shrinking: Presents if the shrinking heuristic will be used in creating the decision boundaries for r-SVM. The Scikit Learn API allows Boolean values. We use both True and False.
- tol: Presents the tolerance as stopping criterion that will be used in creating the decision boundaries for r-SVM. The Scikit Learn API allows multiple integer values. We use 10^{-10}, 1^{-05}, 1^{-02}, and 1.
- decision_function_shape: Presents what shape of the decision function will be created and returned for r-SVM. The Scikit Learn API allows two String values 'ovo', and 'ovr', and 'None' type object. We use all of these three values.

RF: the parameters that we used are:

- n_estimators: Presents the number of trees that will be created in the random forest model. The Scikit Learn API allows multiple integer values. We use 10, 50, 100, and 500.
- criterion: the criterion technique to select a split. The Scikit Learn API allows two String values: 'gini', and 'entropy'. We use both of these values.
- max_features: presents the number of features to consider when making a split. The Scikit Learn API allows multiple values. We use 'auto', 'sqrt', 'log2', and None.
- max_depth: The maximum allowed depth of the decision tree. The Scikit Learn API allows multiple integer values and 'None' type object. We use 5, 15, 50, and 'None'.
- max_leaf_nodes: Presents how many leaves will be used per node when creating the decision tree in a best first fashion. The Scikit Learn API allows multiple integer values and 'None' type object. We use 25, 50, 75, and 'None'.
- Bootstrap: Presents if bootstrapped samples will be used to create the random forests model. The Scikit Learn API allows Boolean values. We use True and False.
- min_samples_split: Presents the minimum number of samples required to split an internal node. The Scikit Learn API allows multiple integer values. We use 1, 25, 50, and 100.
- Oob_score: Presents if out-of-bag samples will be used to create the random forests model. The Scikit Learn API allows Boolean values. We use True and False.
- min_weight_fraction_leaf : Presents the minimum weighted fraction of samples required to be at the leaf nodes of created trees in the random forest. The Scikit Learn API allows floating point values between 0.0 and 0.5. We use 0.0, 0.2, 0.3, and 0.4.

Empirical findings: Feature Selection

Code Metric	**Coefficient**
Number of classes	-1.087×10^{-02}
Lines of code	-2.054×10^{-03}
Methods	2.708×10^{-03}
Duplicated Lines	1.712×10^{-03}
Complexity	2.268×10^{-03}
Class Complexity	7.378×10^{-02}
Function Complexity	***2.306***
Comment lines	-2.770×10^{-03}
Density of Comment Lines	9.980×10^{-02}
Percentage of duplicated lines	$\boldsymbol{-1.599 \times 10^{-01}}$
Files	4.371×10^{-03}
Directories	3.228×10^{-01}
File Complexity	-7.172×10^{-02}
Violations	-6.922×10^{-03}
Duplicated blocks	-8.162×10^{-03}
Duplicated Files	2.702×10^{-01}
Lines	1.202×10^{-03}
Blocker Violations	-1.129×10^{-01}
Critical Violations	3.946×10^{-02}
Major Violations	6.115×10^{-03}
Minor Violations	7.176×10^{-03}

Table 2: Coefficients for Code Metrics Considering Two Levels of Risk

Empirical Findings: Classification models

Following tables present a detailed analysis of the Precision, Recall and F-measure results for all the classification models.

2 Classes												
Class	**precision**				**recall**				**F-Measure**			
	2-Fold		**5-Fold**		**2-Fold**		**5-Fold**		**2-Fold**		**5-Fold**	
	Default	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**
L	0.12	1	0.21	0.68	0.12	0.12	0.35	0.76	0.12	0.21	0.26	0.72
H	0.97	0.97	0.98	0.99	0.97	1	0.96	0.99	0.97	0.99	0.97	0.99
12 Classes												
L8	0	0	0.5	0.3	0	0	0.5	0.5	0	0	0.5	0.37
L7	0.76	0.72	0.77	0.78	0.72	1	0.72	0.9	0.74	0.84	0.75	0.83
L9	0	0	0	0	0	0	0	0	0	0	0	0
L10	0.29	0	0.33	1	0.67	0	0.67	0.33	0.4	0	0.44	0.5
L5	0.25	0	0.28	0.43	0.26	0	0.26	0.22	0.25	0	0.27	0.29
L3	0.08	0	1	1	0.33	0	1	0.67	0.12	0	1	0.8
L2	0.29	0	0.22	0	0.29	0	0.57	0	0.29	0	0.32	0
L0	0	0	0.5	0.38	0	0	0.83	0.5	0	0	0.62	0.43
L4	0	0	0.6	0.5	0	0	0.5	0.5	0	0	0.55	0.5
L6	0.75	0.67	0.38	0.8	0.43	0.57	0.86	0.57	0.55	0.62	0.52	0.67
L11	0.4	0	0.27	0.75	0.33	0	0.5	0.5	0.36	0	0.35	0.6
L1	0.33	0	0.3	1	0.25	0	0.75	0.75	0.29	0	0.43	0.86
13 Classes												
L7	0.74	0.73	0.77	0.76	0.72	0.93	0.7	0.81	0.74	0.82	0.74	0.79
L5	0.24	0.17	0.28	0.28	0.25	0.05	0.25	0.25	0.24	0.07	0.26	0.26
L10	0	0	0	0	0	0	0	0	0	0	0	0
L11	0.17	0.67	0.5	0	0.67	0.67	0.67	0	0.17	0.67	0.57	0
L12	0	0	0.1	0.75	0	0	0.17	0.5	0	0	0.12	0.6
L3	0.08	0	0.5	0.5	0.33	0	1	1	0.08	0	0.67	0.67
L2	0.2	0	0.26	1	0.14	0	0.71	0.14	0.2	0	0.38	0.25
L0	0	0	0.83	0.75	0	0	0.83	0.5	0	0	0.83	0.6

L4	0	0	0.25	1	0	0	0.5	0.17	0	0	0.33	0.29
L6	0	1	0.43	1	0	0.14	0.86	0.71	0	0.25	0.57	0.83
L9	0.43	1	0.33	1	0.6	0.6	0.6	0.6	0.43	0.75	0.43	0.75
L1	0.5	1	0.38	1	0.75	0.5	0.75	0.75	0.5	0.67	0.5	0.86
L8	0	0	0	0	0	0	0	0	0	0	0	0

Table 3 :Detailed CART Results

2 Classes												
Class	**precision**				**recall**				**F-Measure**			
	2-Fold		**5-Fold**		**2-Fold**		**5-Fold**		**2-Fold**		**5-Fold**	
	Default	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**
L	0	1	1	1	0	0.12	0.29	0.47	0	0.21	0.45	0.64
H	0.97	0.97	0.98	0.98	1	1	1	1	0.98	0.99	0.99	0.99
12 Classes												
L8	0	0	1	1	0	0	0.5	0.5	0	0	0.67	0.67
L7	0.75	0.76	0.77	0.79	0.94	0.96	0.94	0.99	0.84	0.85	0.88	0.85
L9	0	0	0	0	0	0	0	0	0	0	0	0
L10	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.67
L5	0.25	0.42	0.43	0.81	0.1	0.13	0.14	0.2	0.15	0.2	0.32	0.21
L3	1	0.75	1	1	1	1	1	1	1	0.86	1	1
L2	0	0	1	1	0	0	0.29	0.29	0	0	0.44	0.44
L0	0	0	1	1	0	0	0.83	0.83	0	0	0.91	0.91
L4	0	0	0.6	0.75	0	0	0.5	0.5	0	0	0.6	0.55
L6	1	1	0.67	1	0.43	0.43	0.86	0.86	0.6	0.6	0.92	0.75
L11	0.5	0.67	0.5	0.67	0.17	0.33	0.17	0.33	0.25	0.44	0.44	0.25
L1	1	1	1	1	0.75	0.75	0.75	0.75	0.86	0.86	0.86	0.86
13 Classes												
L7	0.75	0.76	0.77	0.79	0.92	0.95	0.93	0.99	0.82	0.84	0.85	0.88

L5	0.21	0.31	0.38	0.8	0.09	0.1	0.14	0.19	0.13	0.16	0.21	0.31
L10	0	0	0	0	0	0	0	0	0	0	0	0
L11	0.67	0.67	1	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.8	0.67
L12	0.67	0.75	0.75	0.75	0.33	0.5	0.5	0.5	0.44	0.6	0.6	0.6
L3	1	1	1	1	1	1	1	1	1	1	1	1
L2	0	0	1	1	0	0	0.29	0.29	0	0	0.44	0.44
L0	0	0	1	1	0	0	0.83	0.83	0	0	0.91	0.91
L4	0	0	1	1	0	0	0.5	0.5	0	0	0.67	0.67
L6	1	1	0.67	1	0.43	0.43	0.86	0.86	0.6	0.6	0.75	0.92
L9	1	1	0.75	1	0.6	0.6	0.6	0.6	0.75	0.75	0.67	0.75
L1	1	1	1	1	0.75	0.75	0.75	0.75	0.86	0.86	0.86	0.86
L8	0	0	0	0	0	0	0	0	0	0	0	0

Table 4 :Detailed Random Forests Results

2 Classes												
Class	**precision**				**recall**				**F-Measure**			
	2-Fold		**5-Fold**		**2-Fold**		**5-Fold**		**2-Fold**		**5-Fold**	
	Default	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**
L	0	0	1	1	0	0	0.18	0.18	0	0	0.3	0.3
H	0.97	0.97	0.97	0.97	1	1	1	1	0.98	0.98	0.99	0.99
12 Classes												
L8	0	0	0	0	0	0	0	0	0	0	0	0
L7	0.72	0.71	0.72	0.72	1	0.99	1	1	0.83	0.83	0.84	0.84
L9	0	0	0	0	0	0	0	0	0	0	0	0
L10	0	0	0	0	0	0	0	0	0	0	0	0
L5	0	0	1	1	0	0	0.01	0.01	0	0	0.02	0.02
L3	0	0	0	0	0	0	0	0	0	0	0	0
L2	0	0	0	0	0	0	0	0	0	0	0	0

L0	0	0	1	1	0	0	0.5	0.5	0	0	0.67	0.67
L4	0	0	1	1	0	0	0.33	0.33	0	0	0.5	0.5
L6	0	0	0	0	0	0	0	0	0	0	0	0
L11	0	0	0	0	0	0	0	0	0	0	0	0
L1	0	0	0	0	0	0	0	0	0	0	0	0
13 Classes												
L7	0.72	0.72	0.72	0.72	1	1	1	1	0.83	0.83	0.84	0.84
L5	0	0	1	1	0	0	0.01	0.01	0	0	0.02	0.02
L10	0	0	0	0	0	0	0	0	0	0	0	0
L11	0	0	0	0	0	0	0	0	0	0	0	0
L12	0	0	0	0	0	0	0	0	0	0	0	0
L3	0	0	0	0	0	0	0	0	0	0	0	0
L2	0	0	0	0	0	0	0	0	0	0	0	0
L0	0	0	1	1	0	0	0.5	0.5	0	0	0.67	0.67
L4	0	0	1	1	0	0	0.33	0.33	0	0	0.5	0.5
L6	0	0	0	0	0	0	0	0	0	0	0	0
L9	0	0	0	0	0	0	0	0	0	0	0	0
L1	0	0	0	0	0	0	0	0	0	0	0	0
L8	0	0	0	0	0	0	0	0	0	0	0	0

Table 5 : Detailed SVM Results

2 Classes												
Class	**precision**				**recall**				**F-Measure**			
	2-Fold		**5-Fold**		**2-Fold**		**5-Fold**		**2-Fold**		**5-Fold**	
	Default	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**	**Default**	**Tuned**
L	0	0	1	1	0	0	.06	.29	0	0	.11	.45
H	.97	.97	.97	.98	.99	1	1	1	.98	.98	.99	.99
12 Classes												
L8	0	0	.5	1	0	0	.5	.5	0	0	.5	.67
L7	.75	.73	.76	.75	.84	1	.83	.99	.79	.84	.8	.86

L9	0	0	0	0	0	0	0	0	0	0	0	0
L10	0	1	0	1	0	.67	0	.67	0	.8	0	.8
L5	.33	1	.37	.78	.30	.06	.35	.07	.31	.11	.36	.12
L3	0	0	0	1	0	0	0	.67	0	0	0	.8
L2	0	0	0	1	0	0	0	.29	0	0	0	.44
L0	0	0	.5	1	0	0	.17	.83	0	0	.25	.91
L4	0	0	0	1	0	0	0	.33	0	0	0	.5
L6	0	0	0	1	0	0	0	.29	0	0	0	.44
L11	0	1	0	0	0	.17	0	0	0	.29	0	0
L1	0	0	.17	1	0	0	.25	.5	0	0	.2	.67
13 Classes												
L7	.75	.73	.76	.75	.84	1	.83	.99	.79	.85	.79	.86
L5	.33	1	.36	.78	.3	.06	.36	.07	.32	.11	.36	.12
L10	0	0	0	0	0	0	0	0	0	0	0	0
L11	0	1	0	1	0	.67	0	.67	0	.8	0	.8
L12	.5	1	.75	0	.17	.17	.5	0	.25	.29	.6	0
L3	0	0	0	1	0	0	0	.67	0	0	0	.8
L2	0	0	0	1	0	0	0	.29	0	0	0	.44
L0	0	0	0	1	0	0	0	.83	0	0	0	.91
L4	0	0	0	1	0	0	0	.33	0	0	0	.5
L6	0	0	0	1	0	0	0	.29	0	0	0	.44
L9	0	1	1	1	0	.6	.6	.6	0	.75	.75	.75
L1	0	0	0	1	0	0	0	.5	0	0	0	.67
L8	0	0	0	0	0	0	0	0	0	0	0	0

Table 6 : Detailed KNN Results

2 Classes						
Class	**Precision**		**Recall**		**F-Measure**	
	2-Fold	**5-Fold**	**2-Fold**	**5-Fold**	**2-Fold**	**5-Fold**
L	0	1	0	0.18	0	0.3

H	0.97	0.97	1	1	0.98	0.99
12 Classes						
L8	0	0	0	0	0	0
L7	0.72	0.72	1	1	0.83	0.84
L9	0	0	0	0	0	0
L10	0	0	0	0	0	0
L5	0	1	0	0.01	0	0.02
L3	0	0	0	0	0	0
L2	0	0	0	0	0	0
L0	0	1	0	0.5	0	0.67
L4	0	1	0	0.33	0	0.5
L6	0	0	0	0	0	0
L11	0	0	0	0	0	0
L1	0	0	0	0	0	0
13 Classes						
L7	0.72	0.72	1	1	0.83	0.84
L5	0	1	0	0.01	0	0.02
L10	0	0	0	0	0	0
L11	0	0	0	0	0	0
L12	0	0	0	0	0	0
L3	0	0	0	0	0	0
L2	0	0	0	0	0	0
L0	0	1	0	0.5	0	0.67
L4	0	1	0	0.33	0	0.5
L6	0	0	0	0	0	0
L9	0	0	0	0	0	0
L1	0	0	0	0	0	0
L8	0	0	0	0	0	0

Table 7 :Default Naive Bayes Results