ILLINOIS INSTITUTE OF TECHONOLOGY

CS 584 Machine Learning

Software Defects Binary Classification

Anthony Rodriguez

arodriguez76@hawk.iit.edu

(A20547980)

Deshon Langdon

dlangdon1@hawk.iit.edu

(A20503832)

Dr. Binghui Wang

Submission Date: 6th December 2023

IT IS DECLARED WITH MUTUAL AGREEMENT THAT EACH MEMBER HAS EQUALLY CONTRIBUTED TO THE PROJECT

1. Abstract

The project, "Software Defects Binary Classification using Machine Learning," tackles the challenge of software defects in the software development lifecycle. Software defects can cause system failures, leading to significant disruptions and losses. Therefore, early detection of defects is crucial for preventing failures and ensuring the smooth operation of systems. However, manual identification of defects poses various challenges, such as time-consuming processes, delayed detection, increased costs, and difficulties in resource allocation.

To overcome these challenges, the project proposes implementing various machine learning models to perform binary classification of software defects. The goal is to predict the presence or absence of defects in a given dataset, which contains features related to software, its development process, and its operational environment. The project evaluates the performance of these models based on their ROC AUC Score, Average Training ROC AUC Score, Average Validation ROC AUC Score, Average Accuracy, and Average F1 Score F1 in predicting software defects on a validation set through various K folds. By comparing the models, the project aims to identify the most effective model for early defect detection.

- The expected outcomes include:
- A more efficient and automated defect detection process.
- Leading to reduced manual efforts.
- Early defect identification.
- Cost savings.
- Optimized resource allocation.
- Improved software quality.

The project also addresses scalability issues, making the defect identification and rectification process more feasible as software size and data volume increase. Ultimately, the project aims to enhance the reliability of software systems and provide a better user experience.

2. Introduction

In the intricate world of software development, the presence of defects, or 'bugs,' is an unavoidable reality. These bugs are not just minor hiccups; they can lead to catastrophic system failures, causing significant disruptions and financial losses. The early detection of these defects is therefore crucial in preventing such failures and ensuring the smooth operation of systems. However, manual identification of defects poses various challenges, such as time-consuming processes, delayed detection, increased costs, and difficulties in resource allocation. Therefore, there is a need for an automated and efficient defect detection process that can enhance the quality and reliability of software systems.

This project aims to implement and evaluate various machine-learning models for the binary classification of software defects. The models include Gaussian Naive Bayes, Bernoulli Naive Bayes, Decision Tree Classifier, K-Nearest Neighbors Classifier, Logistic Regression, Random Forest Classifier, Gradient Boosting Classifier, Histogram-based Gradient Boosting Classification Tree, Light Gradient Boosting Machine, Extreme Gradient Boosting, and CatBoost Classifier using dataset "Binary Classification with a Software Defects" take from Kaggle 's Playground Series - Season 3, Episode 23, available on Kaggle, contains 169,605 samples, each with 21 features and a binary label indicating the presence or absence of a defect. The project first implemented baseline models to train them. Then, it used ROC AUC Score, Average

Training ROC AUC Score, Average Validation ROC AUC Score, Average Accuracy, and Average F1 Score as metrics to compare the models' performance. After that, used Hyperparameters such as Gaussian Naive Baye Variance Smoothing for Gaussian Naive Bayes, Alpha for Bernoulli Naive Bayes, Criterion and Max Tree Depth for Decision Tree, Neighbors for K Nearest Neighbor, Solvers, and C values for logistic regression, Max Tree Depth and Learning Rate for His Gradient Boosting, Max features and Max Tree Depth for Random Forest, Max Tree Depth and Max Number of Leaves for Light Gradient Boosting Machine, Max Tree Depth and Learning Rate for Extreme Gradient Boosting and lastly Max Tree Depth for Cat boosting. Then, optimize these models through hyper tunning search that can effectively predict whether software has defects or not.

3. Background and Related Work

A notable study in this field was conducted by researchers at the Software Engineering Institute, who utilized machine learning techniques to classify software defects. They employed a variety of classifiers, including Support Vector Machines (SVM) and Random Forests, to analyze static code metrics and predict defect-prone modules with considerable success. Another influential work was by Jones and Harrold, who explored the use of code churn, complexity metrics, and developer activity as predictors for software defects. Their findings underscored the importance of historical project data in improving the accuracy of defect prediction models. Our approach will involve driving into the heart of Kaggle datasets and finding this endeavor is titled "Binary Classification with a Software Defects Dataset," derived from the Kaggle Playground Series - Season 3, Episode 23. Comprising 169,605 samples, each characterized by 21 features and a binary label indicating the presence or absence of a defect, this dataset serves as the foundation for model training and evaluation.

Research conducted by (J. Abbineni and O. Thalluri, 2018) [11] using Decision Tree and Logistic Regression algorithms was implemented to classify defective modules present in the software so that it helps in improving the quality of the software system. The datasets were taken from the promise data repository, using only accuracy evaluation. Moreover, in this research, the Weka tool and Python idle are used to implement and compare the algorithms. Then, the accuracy of the algorithms is computed and compared. The results show that Decision Tree has higher accuracy than Logistic Regression for most datasets.

Therefore using a similar idea but more additives and a different implementation environment, such as Vs. Code and leveraging a wide range of Python machine learning libraries, our project aimed at where in the project's initial phase involves implementing baseline models mentioned above, which are subsequently trained and evaluated using a set of key metrics. These metrics, including ROC AUC Score, Average Training ROC AUC Score, Average Validation ROC AUC Score, Average Accuracy, and Average F1 Score, provide a nuanced understanding of the models' performance across different dimensions.

Thus, moving beyond the baseline, the project meticulously explores hyperparameters tailored to each specific model. Parameters such as Gaussian Naive Bayes Variance Smoothing, Alpha for Bernoulli Naive Bayes, Criterion, and Max Tree Depth for Decision Tree, Neighbors for K Nearest Neighbor, Solvers and C values for Logistic Regression, Max Tree Depth and Learning Rate for Histogram-based Gradient Boosting, Max Features and Max Tree Depth for Random Forest, Max Tree Depth and Max Number of Leaves for Light Gradient Boosting

Machine, Max Tree Depth and Learning Rate for Extreme Gradient Boosting, and Max Tree Depth for CatBoost are scrutinized.

The project aims to find the best ML model that can effectively predict whether software has defects or not and to analyze the factors that influence the performance of the models; refinements optimization of the models, through hyperparameter tuning search, is employed using various hyperparameters mentioned above. This meticulous optimization process aims to enhance the models' predictive capabilities, ensuring their effectiveness in determining the presence or absence of defects in software.

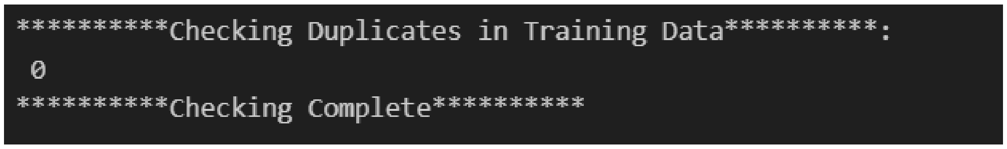
4. Data Exploration

4.1.1 Checking for Missing values and duplicate entries

In the data exploratory phase of building a model using the binary classification with a software defects dataset from Kaggle, the first step was to check for missing values and duplicate entries in the training data. This step is crucial as it ensures the quality and integrity of the data used in the various models implemented later on. Missing values can affect the model's performance, leading to inaccurate predictions or errors during the training process. Therefore, we must identify and handle them appropriately. Regarding the dataset, not checking for missing values could mean overlooking potential defect patterns, leading to less effective defect prediction and mitigation strategies.

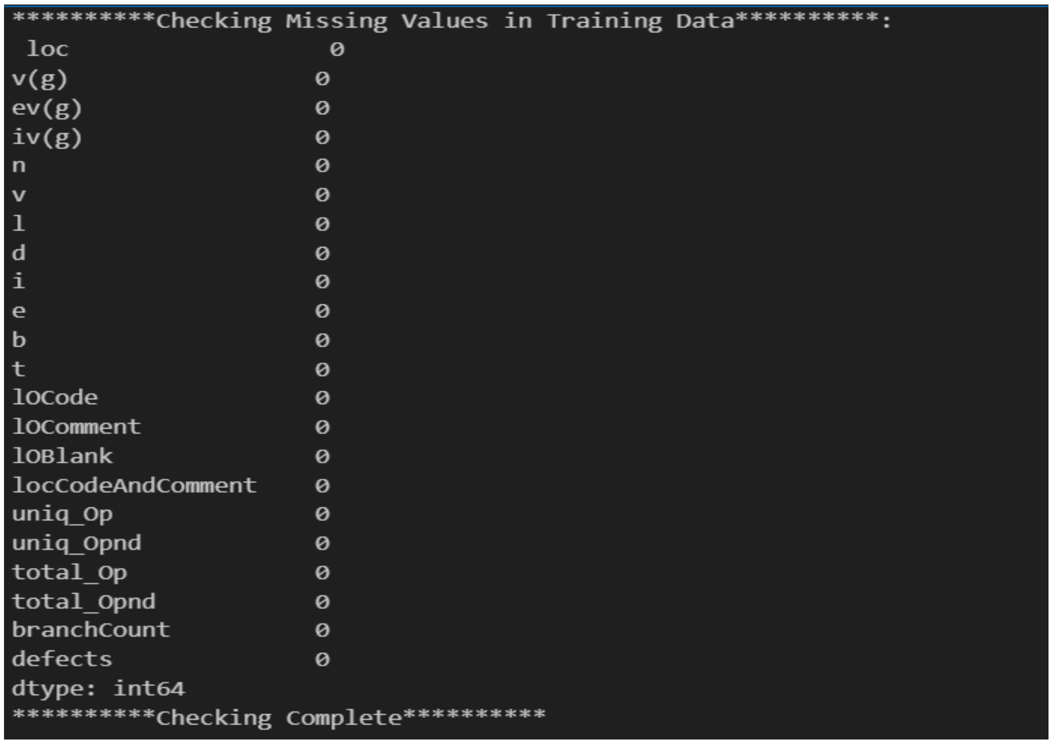
In this case, the isna().sum() function was used to check for missing values in the datasets. The function returns the number of missing values in each dataset column. The result was 0, indicating no missing values in the datasets.

Then, we also checked duplicate entries, which can create bias in the models as they overrepresent certain data points. Therefore, it was vital that if any existed, it needed to be identified and removed to ensure that the various models were trained on a diverse and representative sample of the data. The duplicated().sum() function was used to check for duplicate entries in the datasets. This function returns the number of duplicate rows in the dataset. The result was also 0, indicating no duplicate entries in the datasets. Performing these checks and receiving the results gives 100% confidence that the dataset is clean and ready to be used in the subsequent steps of the model-building process. This not only improves the performance of the model but also ensures that the insights and predictions generated by the model are reliable and accurate.



```
**********Checking Duplicates in Training Data**********:
 0
**********Checking Complete**********
```

Figure 1: Showing results after checking for Duplicates in Training Data.



```
**********Checking Missing Values in Training Data**********:
 loc                  0
v(g)                 0
ev(g)                0
iv(g)                0
n                    0
v                    0
l                    0
d                    0
i                    0
e                    0
b                    0
t                    0
lOCode               0
lOComment            0
lOBlank              0
locCodeAndComment    0
uniq_Op              0
uniq_Opnd            0
total_Op             0
total_Opnd           0
branchCount          0
defects              0
dtype: int64
**********Checking Complete*********
```

Figure 2: Showing results after checking for missing values in Training Data.

4.1.2 Feature and label data distribution

This section of the data exploratory phase deals with an understanding of the structure and distribution of data.

We visualized the distribution of the 21 features in the training dataset using Kernel Density Estimation (KDE) plots, as shown in Figure 3. These plots provided insights into the underlying distribution of each feature, revealing whether they follow a normal distribution, are skewed, or have multiple peaks. Based on the KDE plots, all the predictors showed a positively skewed right-skewed distribution. Therefore, the mean is greater than the median, and the mean overestimates the most common values in a positively skewed distribution, as shown in the label distribution Figure 3 below. There was a great imbalance between non-defects and defects, where 73.33% were defects and 22.7 were not defects. Therefore, since feature distribution had a right-skewed distribution of mostly all the predators, this suggests that many of the software modules don't have defects. In contrast, a high number of software modules don't have defects. In later section 6, these are solved by using cross-validation.

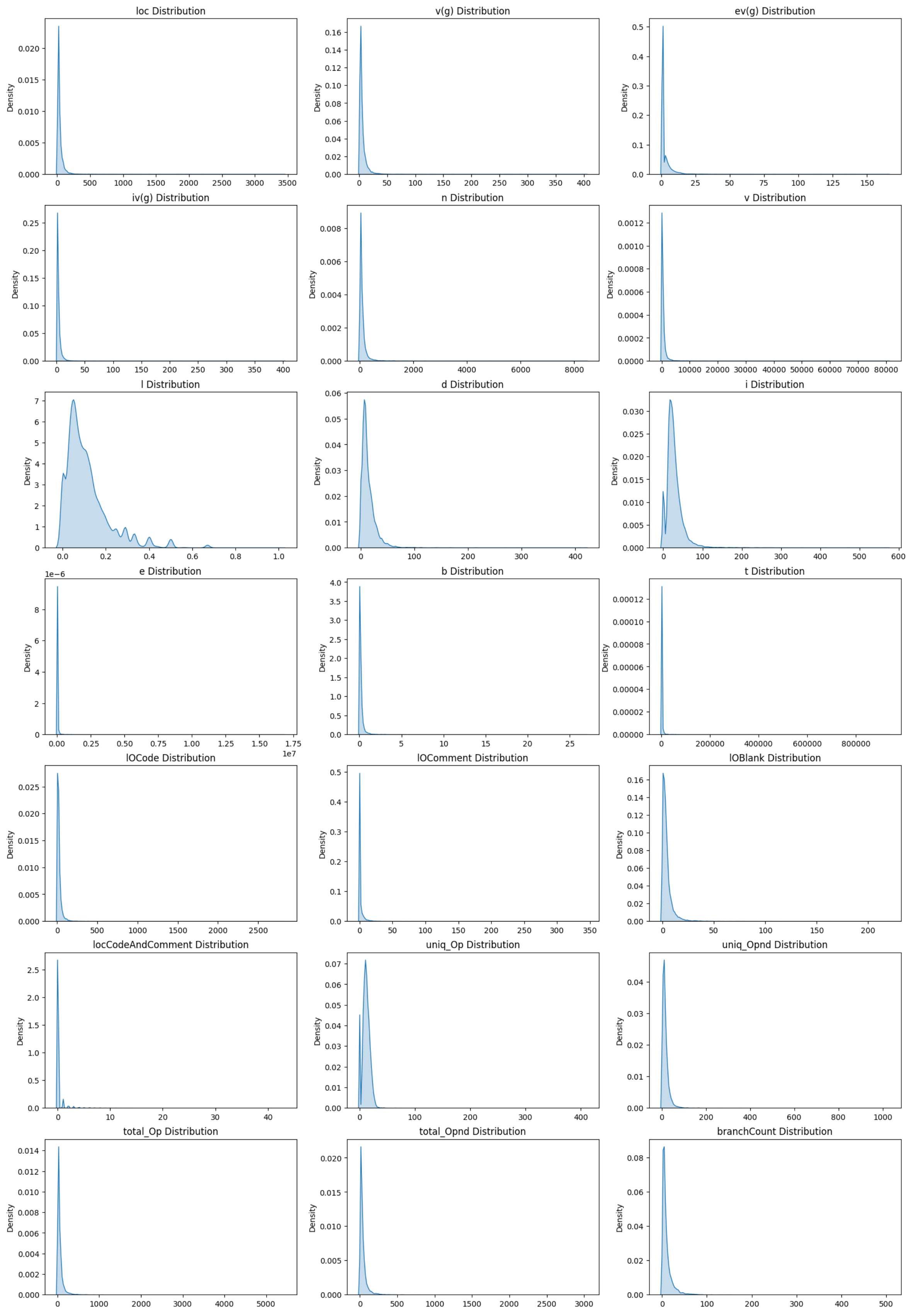

loc Distribution
v(g) Distribution
ev(g) Distribution
iv(g) Distribution
n Distribution
v Distribution
l Distribution
d Distribution
i Distribution
e Distribution
1e−6
1e7
b Distribution
t Distribution
lOCode Distribution
lOComment Distribution
lOBlank Distribution
locCodeAndComment Distribution
uniq_Op Distribution
uniq_Opnd Distribution
total_Op Distribution
total_Opnd Distribution
branchCount Distribution
Density

Figure 3: Showing Kernel Density Estimation (KDE) plots each feature training data distribution.

After that, We visualized the distribution of the label (defects) in the training dataset using just a simple bar chart and pie chart, as shown in Figure 4. The distribution was imbalanced, with a higher percentage of non-defective programs (77.3%) than defective programs (22.7%). This suggests that a high number of software modules don't have defects.

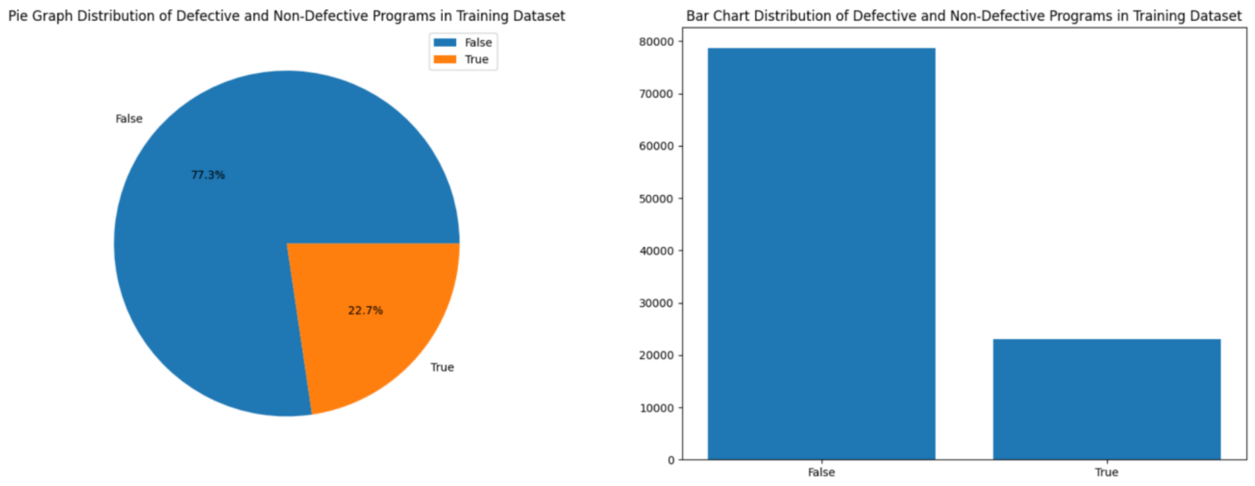


Figure 4: Showing the label (defects) distribution.

Given these distributions, the connection between the feature and label distribution is that the right-skewed feature distribution and the imbalanced label distribution both suggest that a large number of software modules with lower feature values (i.e., those falling in the peak of the right-skewed distribution) are non-defective. Conversely, the software modules with higher feature values (those in the long tail of the right-skewed distribution) are potentially contributing to the smaller proportion of defective modules in the dataset.

4.1.3 Correlation Analysis

In the final stage of the data exploration, it was observed that building on the distributions gathered and similar approaches by the other works on the same dataset, performing correlation analysis gave more insights into the nature of the data and measured the strength and direction of the linear relationship between each feature and the label. Figure 5 shows the correlation matrix among all 21 features.

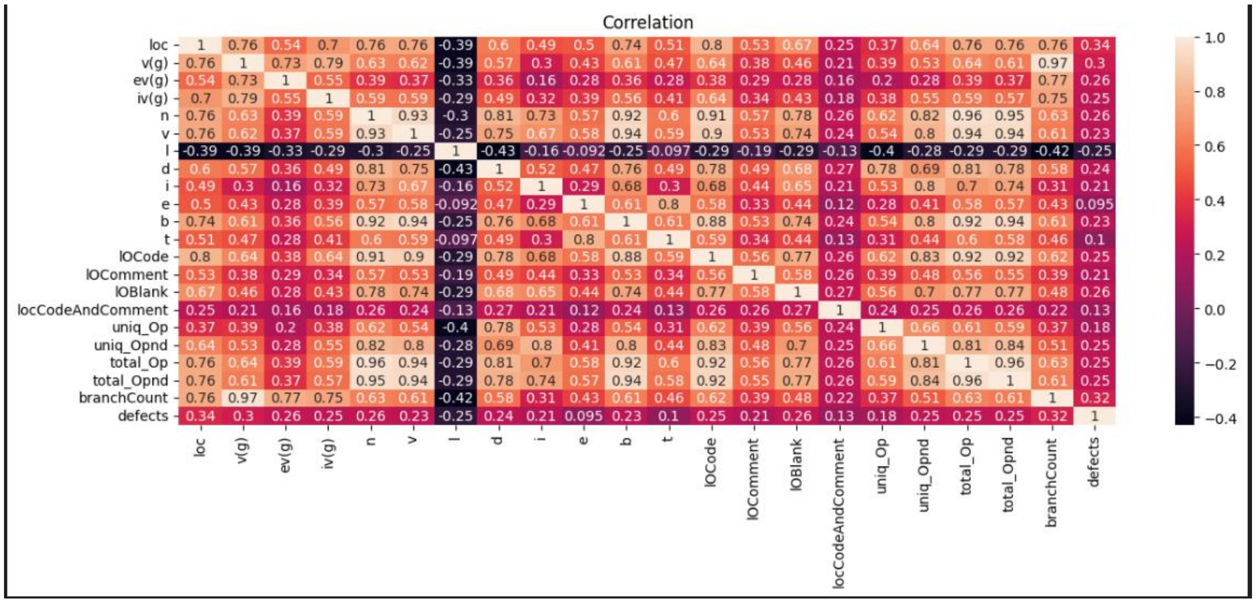


Figure 5 : Showing the correlation matrix of between the features

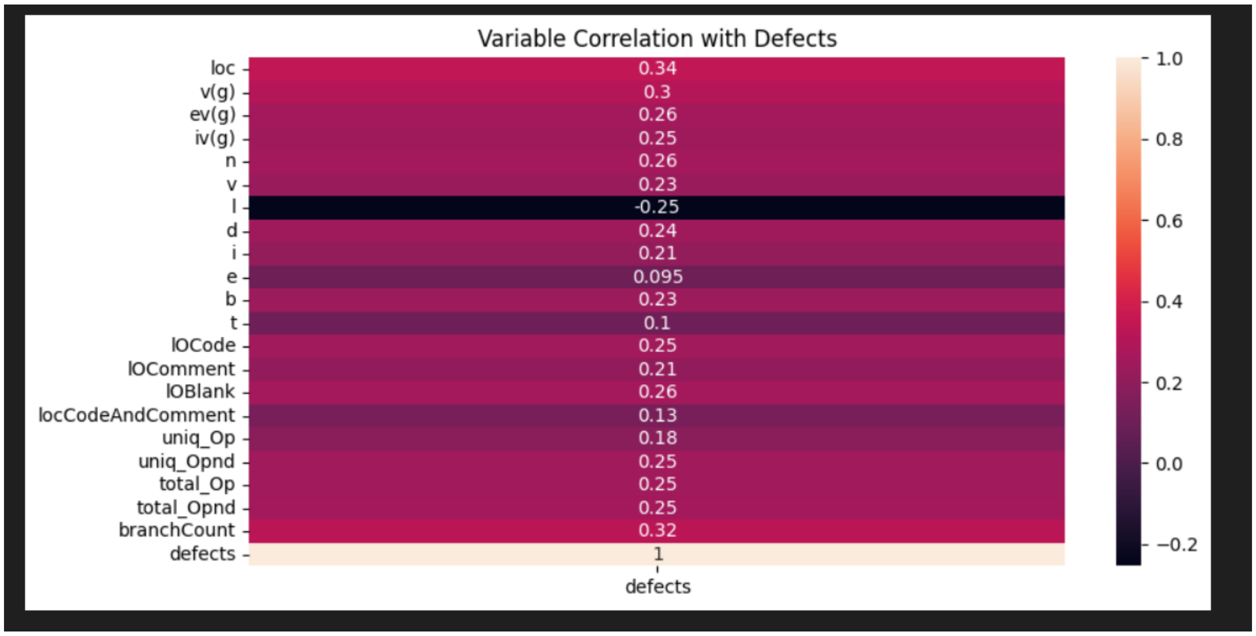


Figure 6: Showing the heat map of distribution.

It was observed that mostly all features were positively correlated, as shown in Figure 3.1 above, and the Y(label) defects are shown in Figure 6 heat map. Some features, such as v(g) and branchCount, had a very high correlation (0.97), which showed they were strongly correlated. Additionally:

1. n to unique_opnd with a correlation of 0.96
2. n to total_Op with a correlation of 0.95
3. v to unique_opnd with a correlation of 0.94
4. v to total_Op with a correlation of 0.94
5. b to n with a correlation of 0.92
6. b to v with a correlation of 0.94

These features have a very high correlation with each other. Even referring back to Figure 3 with the data distribution among the same predictors, the features v(g) and branchCount can see a strong linear relationship because, looking at both distributions, it can be observed that as one increases to peak, so does the other tends to change in a consistent way as well. However, keeping in mind that correlation does not imply causation. Additionally, the correlation among each feature to label (defect) was positively correlated; however, there was no strong correlation. All had moderate or close to no correlation with the label except for feature I, which was the only one that had a negative correlation to the label. As shown in Figure 5, almost all features contributed to Figures 5 and 6. I stood out as the feature that had a negative correlation to all features in the dataset and the label.

Through correlation analysis, it was observed that firstly, the correlation matrix and heat map shown in Figures 3 and 4 explained the variance in the data, opening to the idea that there was multicollinearity. Since multicollinearity is very problematic, the next section of data preprocessing covers how multicollinearity was dealt with.

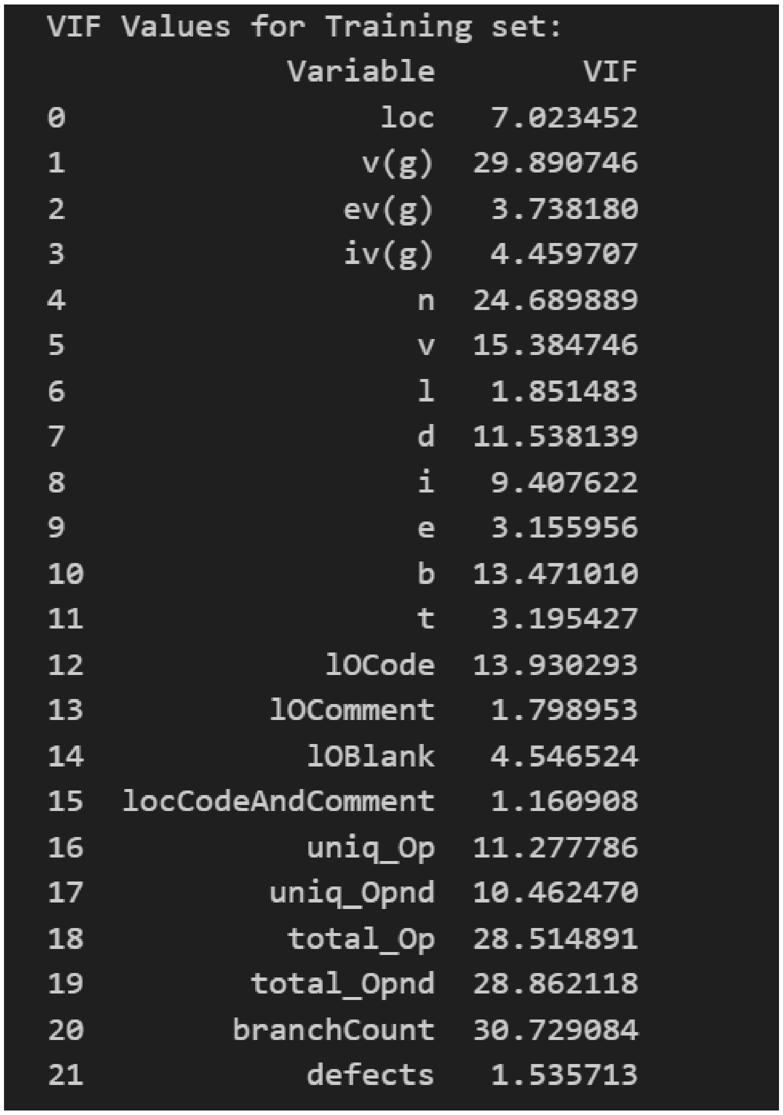
5. Methods

5. 1 Preprocessing

The non-defective and defective software instances in the dataset here required various preprocessing techniques before using them in the Machine Learning models; otherwise, they would have been problematic, which would have skewed the model's performance and affected the stability and interpretability of the model coefficients. Later in the report, we cover what is used to mitigate this factor since it is made known that skewed distributions and class imbalance will have implications on the performance of machine learning if left without doing anything.

5.1.1 Feature Selection

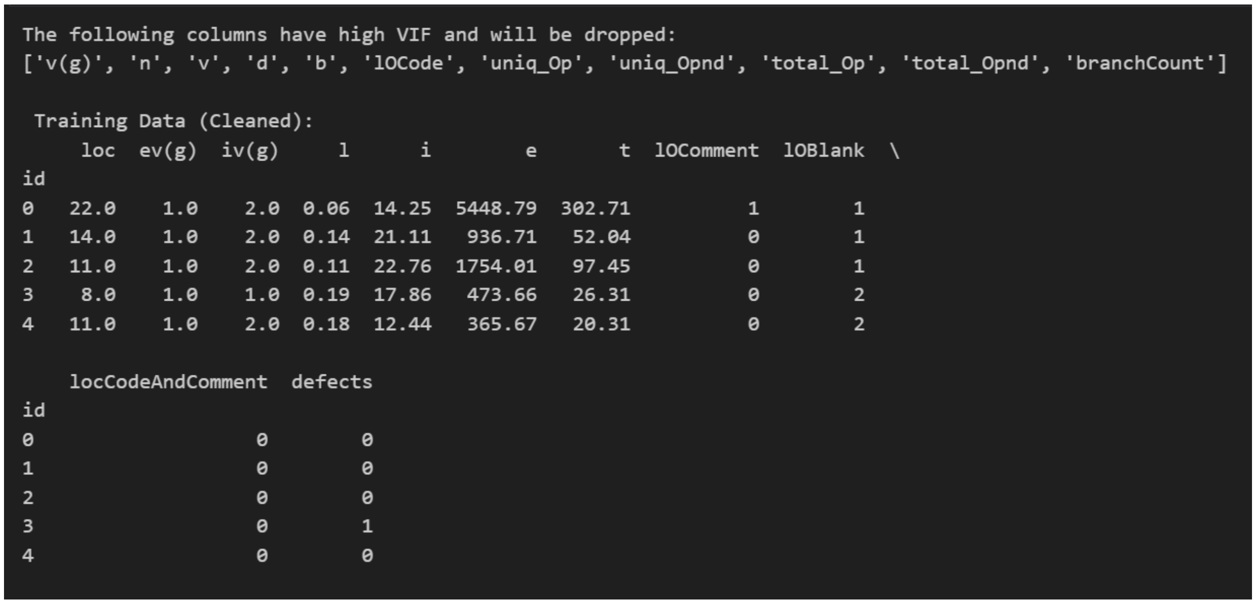
Here, building on the data distribution analysis from the data exploration phase, we learned through correlation analysis that it was observed that the variance in the data was greatly impacted by multicollinearity. In the preprocessing phase, feature selection was focused on removing this multicollinearity among features as possible; through that, the Variance Inflation Factor (VIF) method was used as the main feature extraction method before any data modeling was done. Since the VIF method is a way of measuring how much multicollinearity affects the variance, each feature was treated independently and was regressed on the rest of the independent features, which produced an R-squared value, as shown in Figure 7 that told us how much of the variation in one independent feature is explained by the other independent features. As shown in Figure 7, the VIF value of each indicated how much the coefficient variance for that independent feature is inflated due to multicollinearity. A VIF of 1 means that there is no multicollinearity, and the variance of the coefficient is not affected by the other independent variables. As seen in Figure 7, VIF values for all features were above 1, indicating some multicollinearity and a factor of the VIF values indeed increases the coefficient variance.



```
VIF Values for Training set:
             Variable        VIF
0                 loc   7.023452
1                v(g)  29.890746
2               ev(g)   3.738180
3               iv(g)   4.459707
4                   n  24.689889
5                   v  15.384746
6                   l   1.851483
7                   d  11.538139
8                   i   9.407622
9                   e   3.155956
10                  b  13.471010
11                  t   3.195427
12             lOCode  13.930293
13          lOComment   1.798953
14            lOBlank   4.546524
15  locCodeAndComment   1.160908
16            uniq_Op  11.277786
17          uniq_Opnd  10.462470
18           total_Op  28.514891
19         total_Opnd  28.862118
20        branchCount  30.729084
21            defects   1.535713
```

Figure 7: Showing the VIF values for each feature

As previously mentioned, the v(g) and branchCount had a very high correlation; thus, multicollinearity exists. Therefore, as seen in Figure 7, they have the highest VIF values of 30.729084 for the branchCount feature and 29.890746 for the v(g) feature. Moreover, all the other features with high positive correlation are also reflected with high VIF scores such as "n 24.689889", v 15.384746, total_Op 28.514891, and total_Opnd 28.862118.



```
The following columns have high VIF and will be dropped:
['v(g)', 'n', 'v', 'd', 'b', 'lOCode', 'uniq_Op', 'uniq_Opnd', 'total_Op', 'total_Opnd', 'branchCount']

 Training Data (Cleaned):
     loc  ev(g)  iv(g)     l      i        e       t  lOComment  lOBlank  \
id
0   22.0    1.0    2.0  0.06  14.25  5448.79  302.71          1        1
1   14.0    1.0    2.0  0.14  21.11   936.71   52.04          0        1
2   11.0    1.0    2.0  0.11  22.76  1754.01   97.45          0        1
3    8.0    1.0    1.0  0.19  17.86   473.66   26.31          0        2
4   11.0    1.0    2.0  0.18  12.44   365.67   20.31          0        2

    locCodeAndComment  defects
id
0                   0        0
1                   0        0
2                   0        0
3                   0        1
4                   0        0
```

Figure 8: Showing dropped features and new training dataset features left

Therefore, to reduce the multicollinearity, the standard approach to the VIF method is that if the VIF values are too high, simply drop the features with high values and above the set threshold of 10. As shown in Figure 8, at the top are all the dropped features, and below are the remaining features used in data modeling in the next phase.

5.1.2 Outlier Detection and Standardization

Building upon the previous section, VIF is also sensitive to outliers, as it relies on the R-squared value of the regression of each feature on the rest of the features. Therefore, it was found to be the best fit after visualizing outliers on each feature. The goal is to identify and remove observations that deviate significantly from the rest of the data. Outliers can affect the performance of machine learning algorithms, especially those that use distance measures or weighted sums of inputs, such as k-nearest neighbors, etc., which will discuss the use of the algorithm in a later section. Figure 9 shows the outlier distribution on the optimized dataset. In Figure 9 below, it can be observed that most of the features have values close to zero or zero, except for feature e and feature t. This means that these features have low variability and are mostly constant or zero. This could indicate that these features are not very informative or relevant for the binary classification task, as they do not capture much variation or difference between the software instances.

On the other hand, feature e and feature t have higher values and more variability than the other features. This means these features have more variation and difference between the software instances and may be more informative or relevant for the binary classification task. However, these features also have many outliers, as the boxplot circles show. If not standardized, outliers can affect the performance and accuracy of the machine learning models identified for this project, as they can skew the summary statistics and distributions of the data and make the models less robust and generalizable.

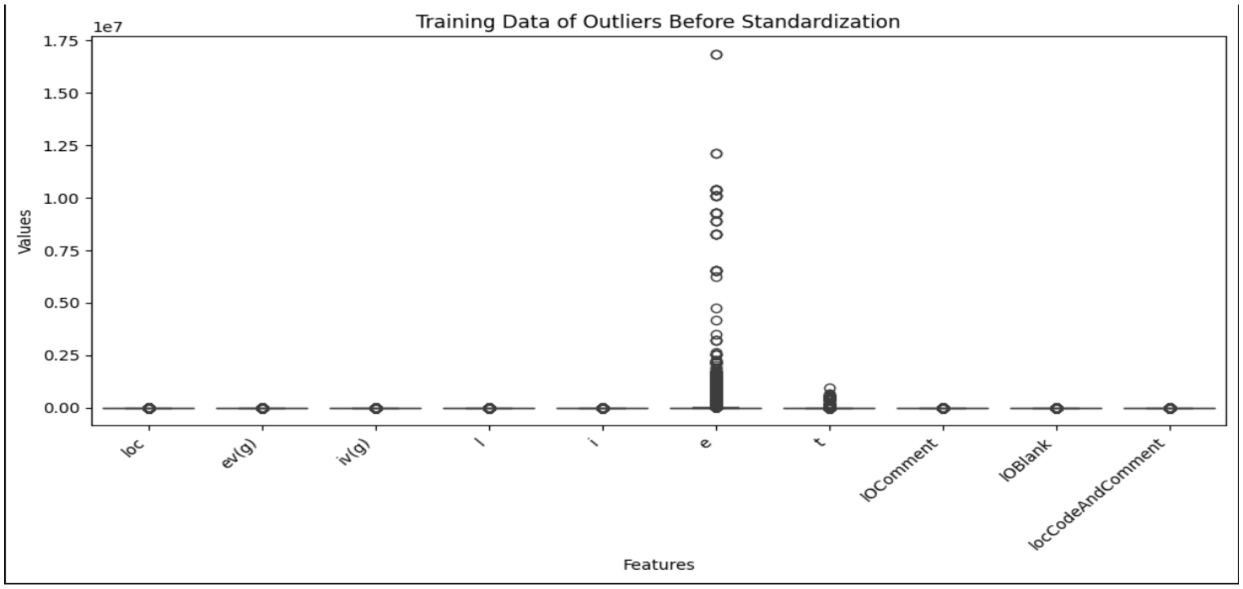


Figure 9: Showing boxplot of features in the training data before standardization.

Given the results in Figure 10 below, the transformation/standardization function. The StandardScaler was applied here to increase their variability and better handle outliers. The StandardScaler transform standardizes the data using the mean and standard deviation. Therefore, the scale and range of the standardized data can be distorted by the outliers, resulting in very large or small values. Looking at Figure 10, it can be observed that the scale of the standardized data is 0 - 80, which is much larger than the original data. This means that the StandardScaler transform inflated the values of the data, especially for the features that have many outliers. As mentioned previously, feature e and feature t have many outliers, as shown by the circles in the boxplots in Figure 10, and they have very large values after standardization. On the other hand, a feature I has few outliers, and it has very small values after standardization.

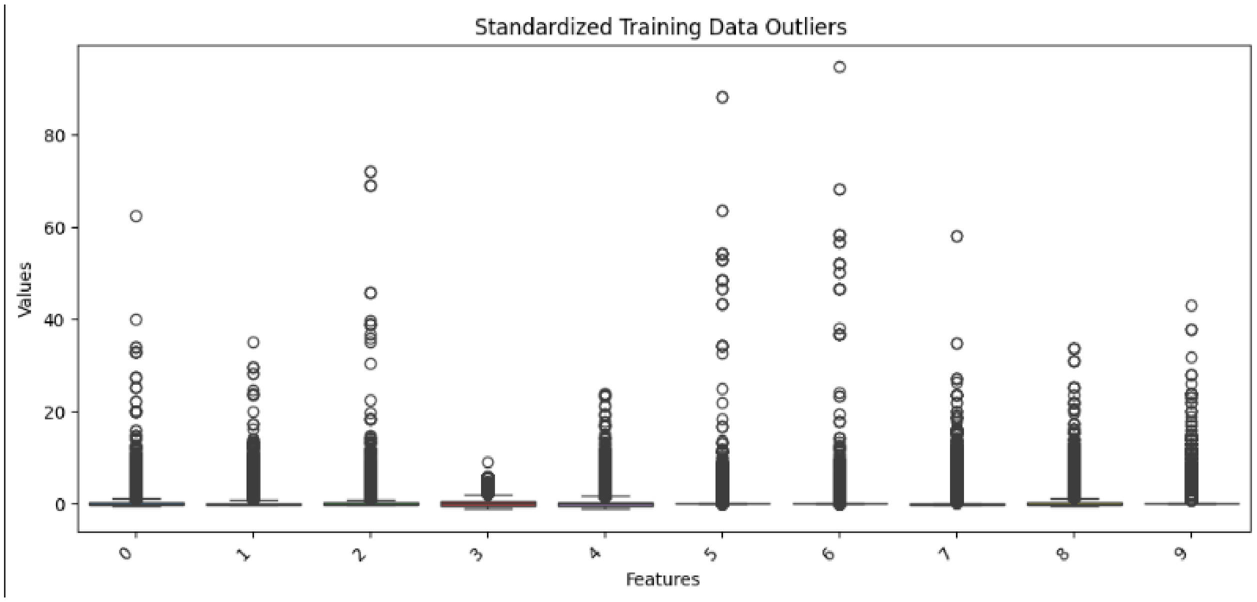


Figure 10: Boxplot of features in the training data after standardization

6.1 Classifier Details

For this project, 11 classifiers were chosen to evaluate the data. Brief explanations of each classifier are provided to explore their differences in operation as they provide different results during the model training.

6.1.1 Naive Bayes

The Naive Bayes Classifiers are based on the Bayes Theorem, a method that allows for the inversion of conditional probabilities. The Bayes Theorem describes the probability of an event based on prior knowledge of conditions potentially related to the event.

The Naive Bayes Classifiers are known as Naive because they assume certain aspects of the dataset to make the classification problems less computationally intensive. First, they assume all of the features equally contribute to the prediction. Second, they assume the features are all independent and unrelated.

The Gaussian Naive Bayes Classifier processes the data, assuming that each data feature is distributed according to the Gaussian distribution.

The Bernoulli Naive Bayes Classifier processes the data according to multivariate Bernoulli Distributions. Thus, each feature is processed as binary-valued. In case the variables are not binary, the instance of the classifier binarizes the input data before processing.

6.1.2 Decision Tree

The Decision Tree Classifier is a classification method that employs a greedy divide-and-conquer method to create the tree. The tree is split until the majority or all data records are classified under the class labels.

6.1.3 K Neighbors

The K Nearest Neighbor Classifier is a classification method that assigns labels to the data points based upon majority voting, where each data point is labeled according to the label of the majority of data points found to be closely related/represented to them.

6.1.4 Logistic Regression

The Logistic Regression Classifier is a classification method that estimates the probability of a classification outcome based on data with features assumed to be independent.

The Logistic Regression fits a logistic sigmoid function predicting binary values, assigning each data point a classification label based on a threshold value.

6.1.5 Random Forest

The Random Forest Classifier is a feature bagging ensemble method composed of decision trees where each tree in the ensemble is made of a data sample drawn from the training data set. The method then trains each tree independently. The predicted class is then outputted by the categorical variables with the majority of votes.

6.1.6 Gradient Boosting

The Gradient Boosting Classifier is a boosting ensemble method based on the gradient descent algorithm. The method iteratively adds predictors to the ensemble to correct the errors of the existing predictors by training on the residual errors of the existing predictor.

6.1.7 Histogram Gradient Boosting

The Histogram Gradient Boosting Classifier is a boosting ensemble method based upon the gradient boosting classifier. The Histogram Gradient Boosting Classifier significantly speeds up the gradient boosting method by binning the input data, speeding up the construction of the decision trees.

6.1.8 Light Gradient Boosting Machine

The Light Gradient Boosting Machine Classifier is a boosting ensemble method based on the histogram gradient boosting classifier. The implementation of this classifier focuses on two separate methods: gradient-based one-side sampling and exclusive feature bundling. Gradient-based one-sided sampling is a method that focuses on training examples that have a larger gradient. Exclusive feature bundling is an automatic feature selection method focused on bundling sparse mutually exclusive features.

6.1.9 Extreme Gradient Boosting

The Extreme Gradient Boosting Classifier is a boosting ensemble method based on the histogram gradient boosting classifier. Optimized with a focus on speed, the classifier has several features, some allowing missing data through the maximization of gain and using weighted quantiles to find the best node splits instead of analyzing all candidates.

6.1.10 Cat Boosting

The Cat Boosting Classifier is a boosting ensemble method based upon the histogram gradient boosting classifier. The implementation of this classifier focuses on forming symmetric decision trees. In addition, the classifier performs well with categorical datasets by focusing on encoding categorical features based on the label columns.

7. Results/Discussion

7.1 Training Procedure

Cross Validation

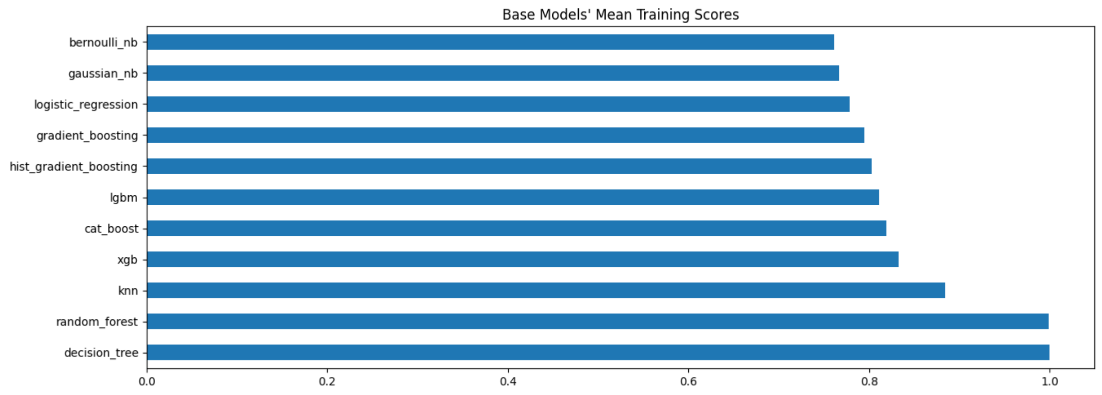
Once data feature selection and standardization were completed, a custom cross-validation process was created to evaluate each classifier and its performance. The repeated stratified k-fold cross-validation method was defined with five folds and ten repeats to ensure that during cross-validation, each fold contains the same proportion of input data with defect and non-defect labels.

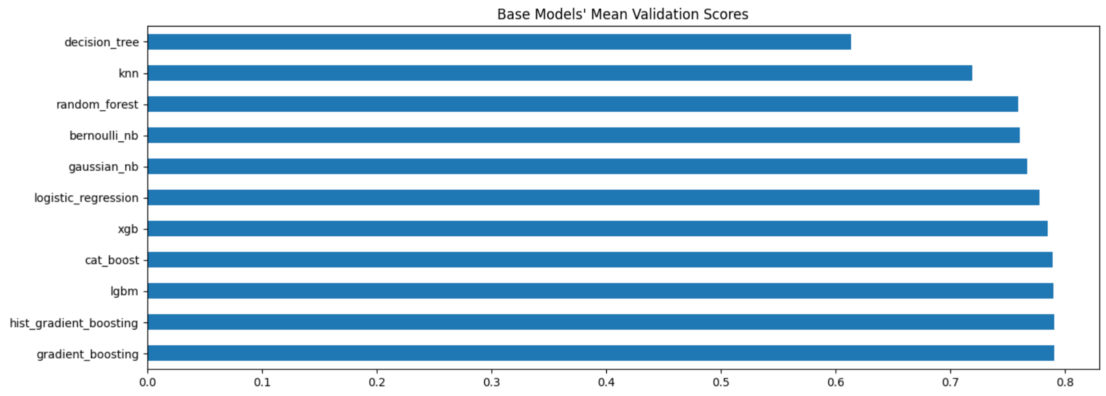
During evaluation, four metrics were analyzed, the average training ROC AUC score, average validation ROC AUC score, average accuracy, and average F1 score. The ROC AUC score is the area under the receiver operating characteristic curve, which plots the true positive rate against the false positive rate at varying classification thresholds. The ROC AUC score details how well a classifier can distinguish between the positive and negative classes. Accuracy describes the percentage of observations that are correctly classified. The F1 score is the harmonic mean of the precision and recall metrics to increase the F1 score is to maximize both metrics. Precision is a metric that measures the ratio of correct positive class predictions.

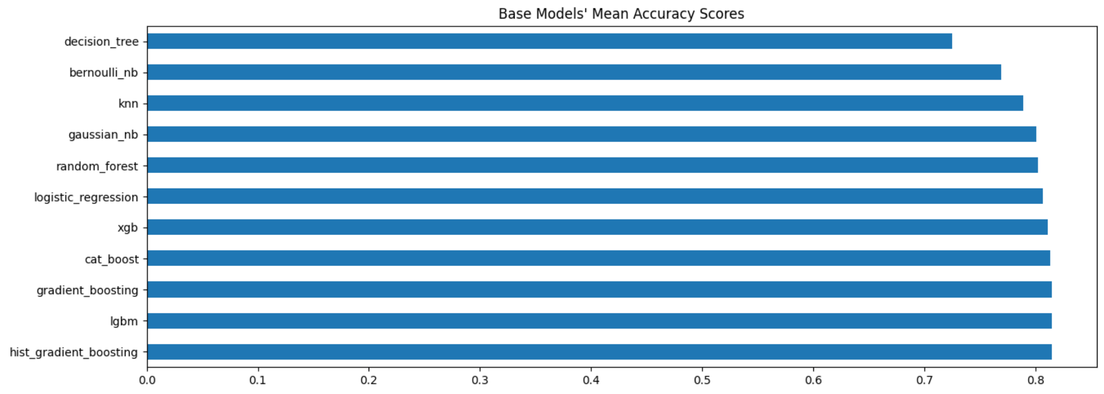
7.2 Training Results

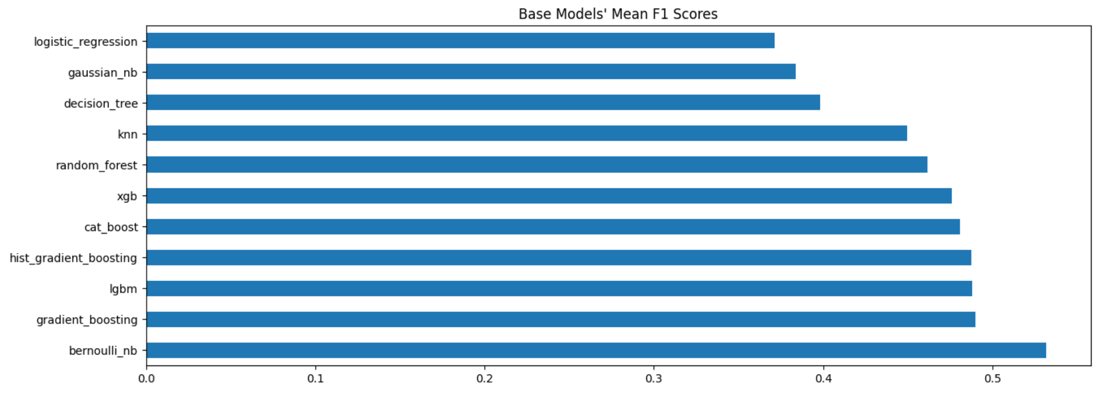
7.2.1 Base Models

Initially, cross-validation was performed on the 11 classifiers without modifying the model parameters beyond adding a seed to ensure consistent results after training. The results of cross-validation are displayed below. The average training ROC AUC scores range between .76 and .99. From these scores, there is an inclination to believe there is an aspect of overfitting in the training of the models. For the average validation ROC AUC, the scores range from .61 to .79. The average accuracy range for the models is .72 to .81. Although accuracy may not accurately represent each classifier's performance since the distribution has a 77:23 skew. Average F1 scores were disappointing, with a range from .37 to .53.









Figures 11 -14: Showing the Base Model Average Training/Validation ROC AUC, Accuracy, and F1 Score Bar Charts

Model Name	**Ave. Training ROC AUC Score**	**Ave. Validation ROC AUC Score**	**Ave. Accuracy**	**Ave. F1 Score**
Gaussian NB	0.7671589	0.7671437	0.8006672	0.3837027
Bernoulli NB	0.760931	0.7608771	0.7692737	0.5317257
Decision Tree	0.9995711	0.6138554	0.7248548	0.3985115
KNN	0.884046	0.7194356	0.7890225	0.4495136
Logistic Regression	0.7781648	0.7780576	0.8067461	0.3711798
Random Forest	0.9985411	0.759114	0.8024754	0.461492
Gradient Boosting	0.7948394	0.7906932	0.8146203	0.4898802
Hist Gradient Boosting	0.8028327	0.7906572	0.8147686	0.4876493
LGBM	0.810897	0.7903612	0.8147549	0.4881741
XGB	0.8323212	0.7849803	0.8113244	0.4759125
Cat Boost	0.8193732	0.7892677	0.8130283	0.4809887

Figure 15: Showing Average Training/Validation ROC AUC, Accuracy, and F1 Scores for All Base Model Classifiers

7.2.2 Classifier Optimization

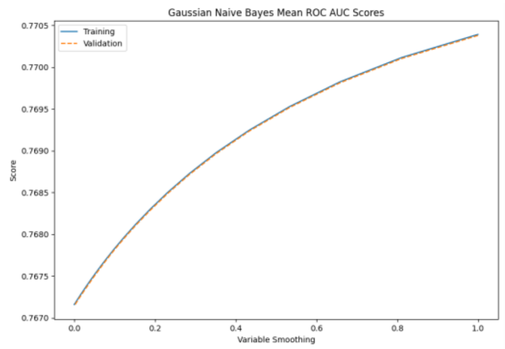
To improve each model and the results of their metrics, each classifier was independently cross-validated to tune their hyperparameters. They were each uniquely tuned based on the available hyperparameters, computational complexity, and the overall effects of the hyperparameter modification.

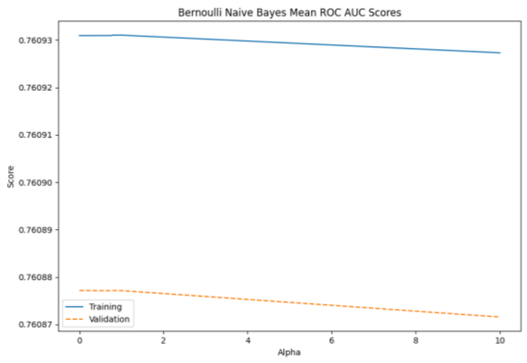
For each of the 11 classifiers, a model with specific hyperparameter values was selected based on the average validation ROC AUC score. The validation ROC AUC represents how well the model can distinguish between classes. Overall, it was believed that the validation ROC AUC would best represent each model's performance with unknown testing data. The hyperparameters modified for each classifier are displayed in the table below. In addition, the mean validation ROC AUC score plots for each model are included below.

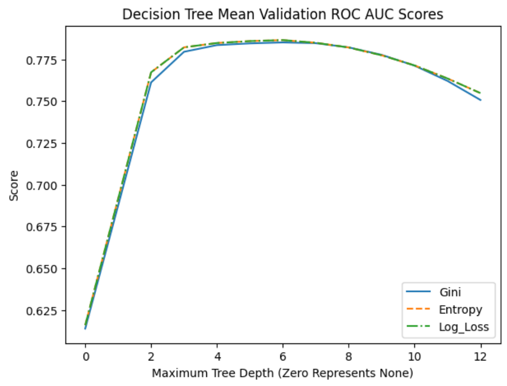
Model	**Hyper Parameters Tuned**
Gaussian Naive Bayes	● Variance Smoothing
Bernoulli Naive Bayes	● Alpha
Decision Tree	● Criterion ● Max Tree Depth
K Nearest Neighbor	● Neighbors
Logistic Regression	● Solvers ● C Values
Random Forest	● Max Tree Depth ● Max Features
Gradient Boosting	● Max Tree Depth ● Learning Rate
Hist Gradient Boosting	● Max Tree Depth ● Learning Rate

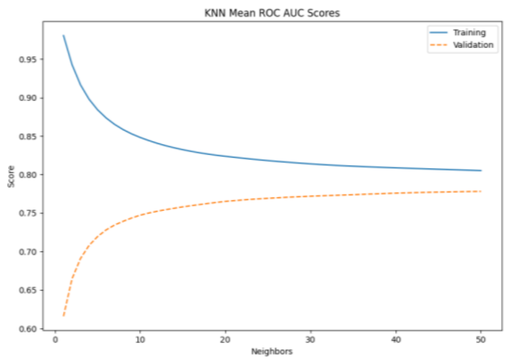
Light Gradient Boosting Machine	● Max Tree Depth ● Max Number of Leaves
Extreme Gradient Boosting	● Max Tree Depth ● Learning Rate
Cat Boosting	● Max Tree Depth

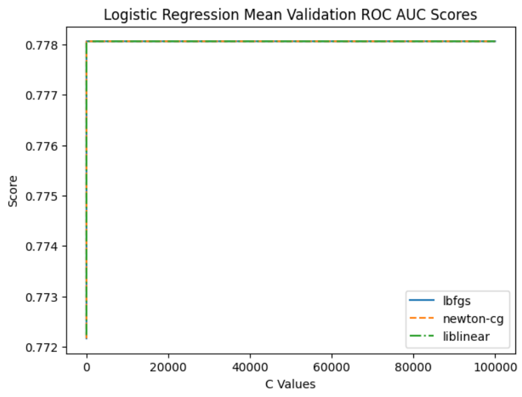
Figure 16: Showing Hyperparameters Tuned for Each Classifier

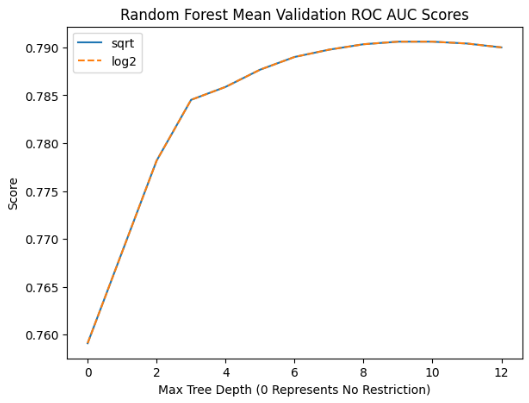


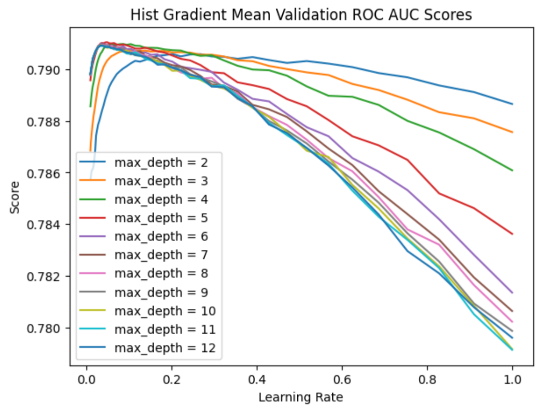


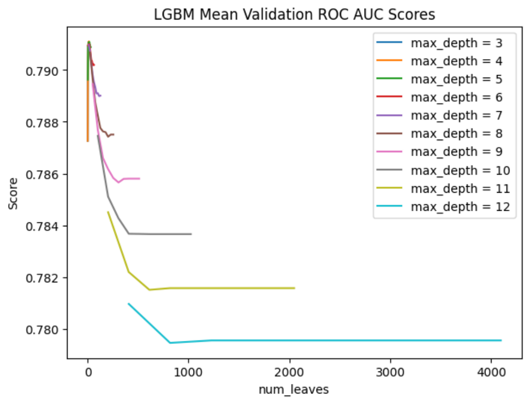


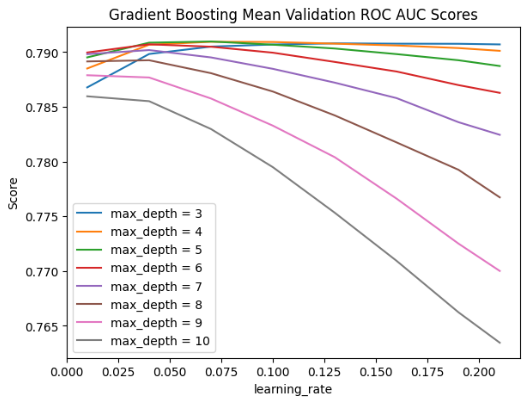


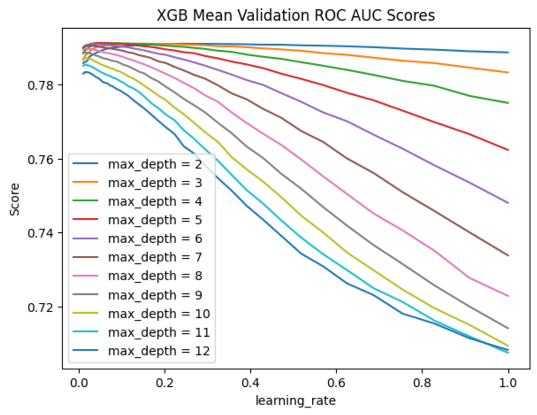


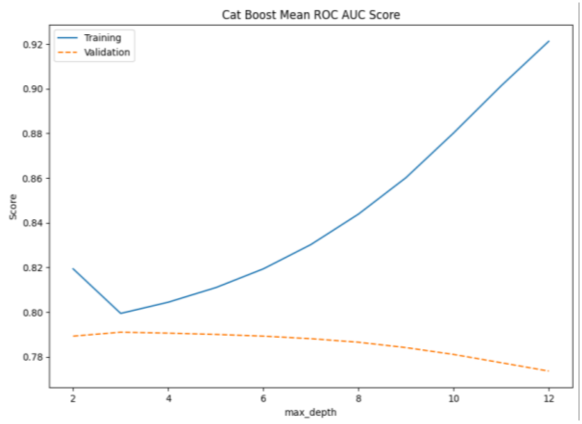












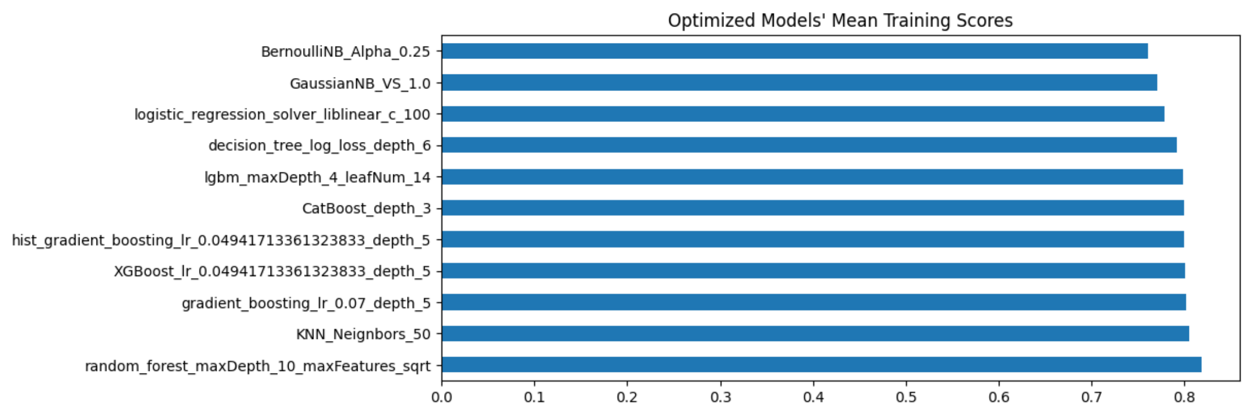
Figures 17- 27: Showing Hyperparameter Tuning Classifier Average Validation ROC AUC Scores

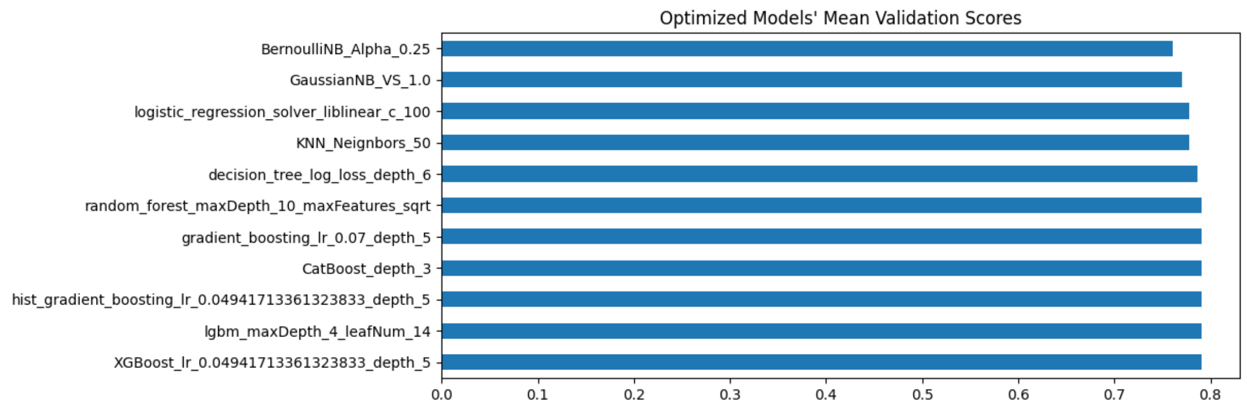
7.3 Evaluation Results

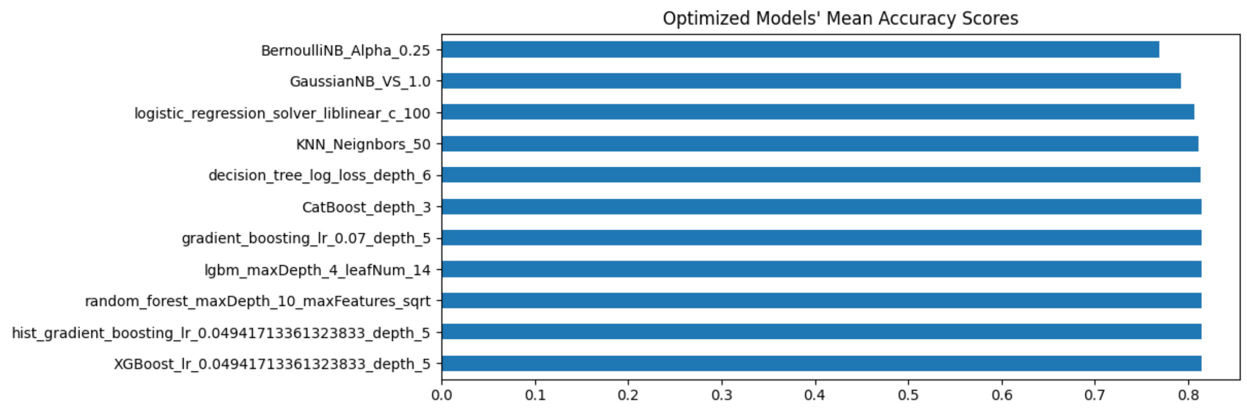
Evaluating Optimized Models

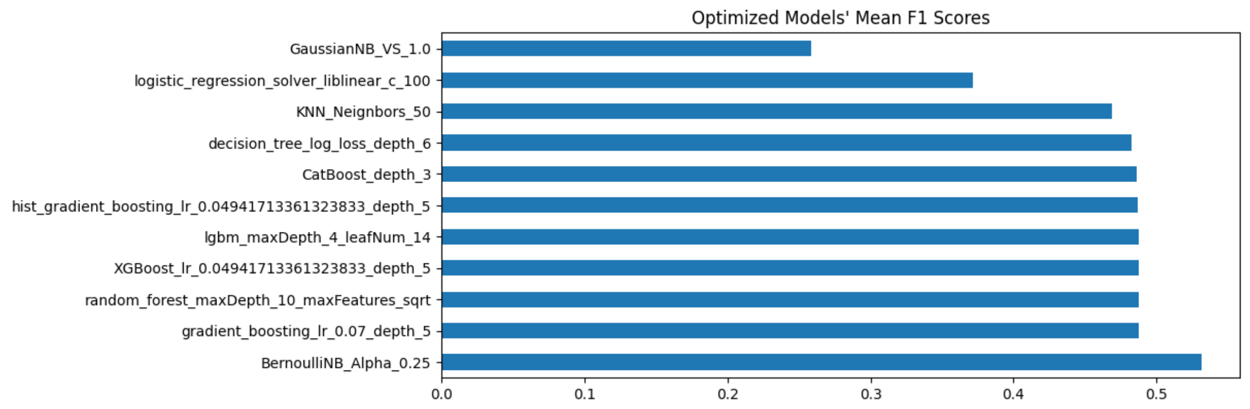
Upon selecting the models from each classifier with the largest mean validation ROC AUC scores, the models were cross-validated once more and evaluated. The average training ROC AUC scores for the optimized models range from .76 to .81. The top end of the range was decreased by .18, dramatically reducing the overfitting of the models. The average accuracy range is from .76 to .81, a slight improvement over the base model range of .72 to .81. The average F1 score range is from .25 to .53, which is a performance decrease from the base model average F1 range of .37 to .53. Now the average validation ROC AUC score range seemed to narrow up to between .76 and .79.

Overall, the optimization of the models seemed to have mainly narrowed the ranges of the metrics. The average F1 score range saw a .12 decrease from the lower end range, which indicates a decline in recall, precision, or both. Overall, the model optimized with preference to the average validation ROC AUC scores did not improve as much as hoped. Instead, there may have been an overall decrease in many of the models. Models like logistic regression slightly improved, but this appears to be the exception. The results of the optimized model cross-validation are below.


Optimized Models' Mean Training Scores
BernoulliNB_Alpha_0.25
GaussianNB_VS_1.0
logistic_regression_solver_liblinear_c_100
decision_tree_log_loss_depth_6
lgbm_maxDepth_4_leafNum_14
CatBoost_depth_3
hist_gradient_boosting_lr_0.04941713361323833_depth_5
XGBoost_lr_0.04941713361323833_depth_5
gradient_boosting_lr_0.07_depth_5
KNN_Neignbors_50
random_forest_maxDepth_10_maxFeatures_sqrt
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8


Optimized Models' Mean Validation Scores
BernoulliNB_Alpha_0.25
GaussianNB_VS_1.0
logistic_regression_solver_liblinear_c_100
KNN_Neignbors_50
decision_tree_log_loss_depth_6
random_forest_maxDepth_10_maxFeatures_sqrt
gradient_boosting_lr_0.07_depth_5
CatBoost_depth_3
hist_gradient_boosting_lr_0.04941713361323833_depth_5
lgbm_maxDepth_4_leafNum_14
XGBoost_lr_0.04941713361323833_depth_5
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8


Optimized Models' Mean Accuracy Scores
BernoulliNB_Alpha_0.25
GaussianNB_VS_1.0
logistic_regression_solver_liblinear_c_100
KNN_Neignbors_50
decision_tree_log_loss_depth_6
CatBoost_depth_3
gradient_boosting_lr_0.07_depth_5
lgbm_maxDepth_4_leafNum_14
random_forest_maxDepth_10_maxFeatures_sqrt
hist_gradient_boosting_lr_0.04941713361323833_depth_5
XGBoost_lr_0.04941713361323833_depth_5
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8



Figures 28 -31: Showing Optimized Model Average Training/Validation ROC AUC, Accuracy, and F1 Score Bar Charts

Model Name & Hyperparameter Values	Ave. Training ROC AUC Score	Ave. Validation ROC AUC Score	Ave. Accuracy	Ave. F1 Score
GaussianNB Variable Smoothing: 1.0	0.7703925	0.7703802	0.7925759	0.2587808
BernoulliNB Alpha: 0.25	0.7609309	0.7608771	0.7692737	0.5317257
Decision Tree Criterion: log loss Max Depth: 6	0.7916948	0.7865117	0.8135423	0.4824395
KNN Neighbors: 50	0.8051304	0.7781011	0.8117106	0.4686465
Logistic Regression Solver: liblinear C Value: 100	0.7781662	0.7780591	0.80675	0.3712207

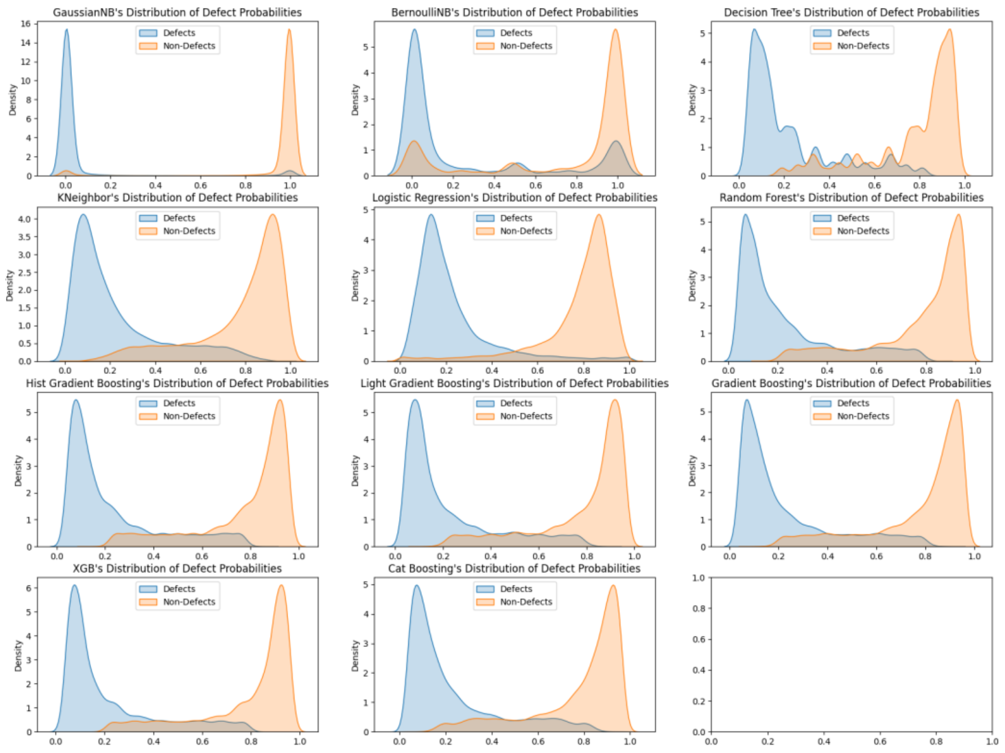
Random Forest Max Depth: 10 Max Features: sqrt	0.8188424	0.790596	0.8148463	0.4876912
Hist Gradient Boosting Learning Rate: 0.04941713361323833 Max Depth: 5	0.7995138	0.7910308	0.8148699	0.4870412
LGBM Max Depth: 4 Max Leaf Number: 14	0.7983258	0.7911049	0.8146743	0.4876798
Gradient Boosting Learning Rate: 0.07 Max Depth: 5	0.8023741	0.7909673	0.8146645	0.4877596
XGBoost Learning Rate: 0.04941713361323833 Max Depth: 5	0.800319	0.7911931	0.814921	0.4876838
CatBoost Max Depth: 3	0.7994325	0.791019	0.8144041	0.4859718

Figure 32: Showing the Average Training/Validation ROC AUC, Accuracy, and F1 Scores for All Optimized Model Classifiers

7.4 Labelling Test Data

Testing Data

The final aspect of this project is analyzing the testing data with the validation of ROC AUC-optimized models. Since the testing data was provided for a Kaggle competition, the actual labels are unknown. Instead, the distribution of defect predictions was calculated for each model. The observation made on the distribution plots is that the ensemble models all appear to have similar distributions. The kernel density estimate plots are displayed below.



Figures 33- 43: Showing Distribution of Defect Predictions for Each Optimized Classifier

8. Conclusion

In conclusion, it was found that for the overall base model classifiers, the average F1 scores appear to be rather low, ranging between .37 and .53. For the base models, the ensemble classifiers and Bernoulli Naive Bayes performed better based upon the average F1 and average validation ROC AUC scores. The model optimization based upon validation ROC AUC did not appear to make much of an improvement, if any. The optimized classifiers that performed best appeared to be the ensembles, all with similar metric results. The lack of improvement through optimizing the average validation ROC AUC may be due to how imbalanced the training data was, skewing the results.

In the future, the group would like to explore the data present in the data sets further before continuing to cross-validate and test the classifier models. Exploring other pre-processing methods may be beneficial for each model to improve the results. The metrics used to evaluate the model would also need reevaluation since the optimization of the models did not yield the expected improvements to the results. In addition, based on the results of the cross-evaluated classifiers, the exploration of a custom classifier ensemble may create an improved classifier for the data.

9. GitHub Link for Project

Project code and data are available at

https://github.com/dlangdon1/Software-Defects-classification-Detection-.git

10. Contributions

Anthony Rodriguez

- Worked on the report.
- Cross Validation
- Training base models
- Classifier Optimization
- Evaluating Optimized Models
- Labelling Test Data

Deshon Langdon

- Worked on the report.
- Data Exploration
- Feature and label data distribution
- Correlation and heat map analysis and heat
- Preprocessing of data includes Feature Selection (VIF method)
- Outlier Detection and Standardization

11. References

Dataset:

https://www.kaggle.com/competitions/aptos2019-blindness-detection/dat

[1] https://www.ibm.com/topics/naive-bayes

[2] https://www.ibm.com/topics/decision-trees

[3] https://www.ibm.com/topics/knn

[4] https://www.ibm.com/topics/logistic-regression

[5] https://www.ibm.com/topics/random-forest

[6] https://www.ibm.com/topics/boosting

[7] https://machinelearningmastery.com/histogram-based-gradient-boosting-ensembles/

[8] https://machinelearningmastery.com/light-gradient-boosted-machine-lightgbm-ensemble/

[9] https://www.geeksforgeeks.org/ml-xgboost-extreme-gradient-boosting/

[10]https://www.geeksforgeeks.org/gradientboosting-vs-adaboost-vs-xgboost-vs-catboost-vs-lightgbm/

[11] (J. Abbineni and O. Thalluri, 2018)