**Prediction of Software defect from the Code**

Akhilesh Potturi – Z1942158

Nikhitha Jetling - Z1924282

**Abstract**

One of the most well-known study areas in computer science is software defect prediction. It aims to find defects occurring from the function level code so it can be used to better prioritize software quality assurance work. We have gone through different methods for predicting the software defect from the code, applied different balancing techniques for imbalance data and removed multicollinearity from the data which helped in giving better results and the best model we have trained is able to predict the defect with 99% Accuracy, 100 % Recall, 98% Precision and 0.99 ROC AUC Score.

**Keywords**: Defect prediction, Principal Component Analysis, Balancing Data, Multicollinearity

**Introduction**

One of the most active fields of study in software engineering is software defect prediction, which is crucial to software quality assurance. The difficulty in producing high quality, affordable, and maintainable software as well as the likelihood of producing software errors have increased due to the software's growing complexity and dependency. Software defects frequently lead to unexpected or inaccurate outputs and behaviors. The negative consequences of the software defect is it increases developer labor costs, Customer Attrition, Loss of potential customers, Loss of Investor Trust, Opportunity loss and suspended growth. According to the Consortium for Information and Software Quality, US businesses lost $2.08 trillion in 2020 as a result of poor software quality. It’s always better to prevent software defects than fixing them.

There isn’t any perfect system which is able to find the defect before it’s occurance. Therefore, we are trying to explore and try to find the best way to solve this issue.

**Literature Review**

Predicting defects is a very important and necessary task. Li[2] has mentioned in his paper about the importance of software defect prediction and said that by identifying defect-prone modules before testing, defect predictors help software engineers efficiently allocate their limited testing and maintenance resources. This lowers costs and increases software quality. Software instance classification as defective or non-defective can be used to anticipate software defects at the method, class, file, change, or package level. Various software metrics and historical defect data gathered from past releases of the same project, or even other projects can be used to build the defect prediction model. Such a model will be developed to determine whether a given piece of software is flawed or not.

**Dataset**

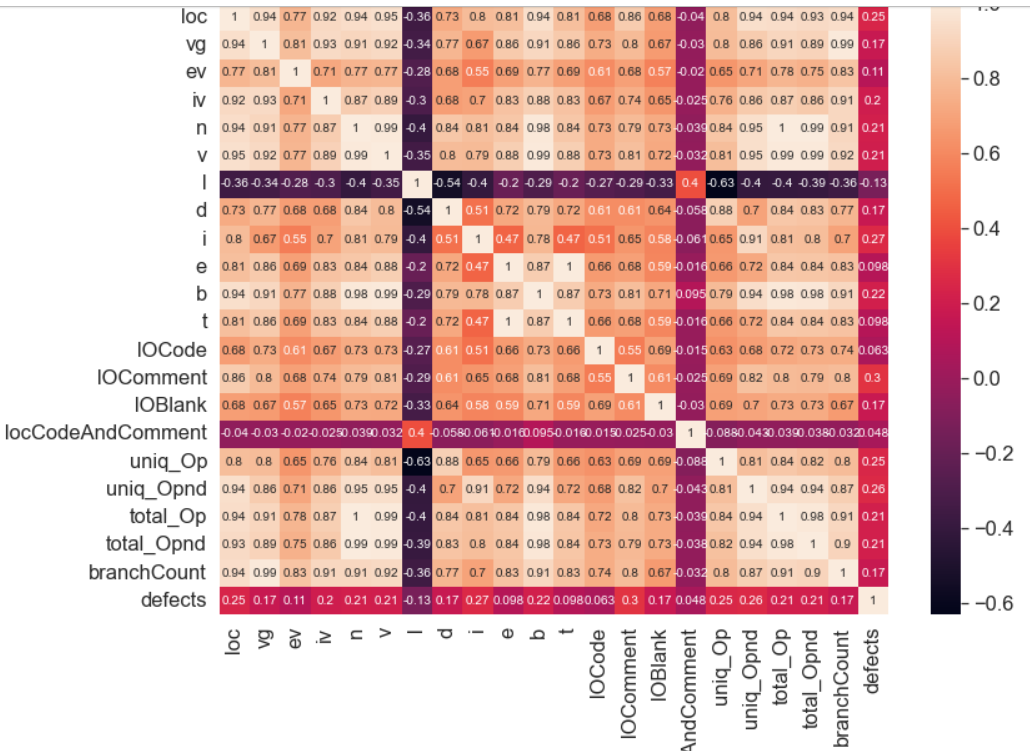
The dataset we have used is Software defect prediction dataset which is a PROMISE Software Engineering Repository data set that was made publicly available in an effort to promote repeatable, verifiable, debatable, and/or improved software engineering predictive models. This dataset is in the form of Attribute-Relation File Format (Arff) format and consists of 22 features and 498 entries. An Arff file is an ASCII text file that describes a list of instances sharing a set of attributes. The first 21 attributes are independent variables and the last attribute defects is the dependent variable. This is a classification dataset and it has two classes True and False. The number of True classes is 49 and the number of False classes is 449. Below is the list of all the attributes in the dataset.

| | 1. loc | McCabe's line count of code | | --- | --- | | 2. v(g) | McCabe "cyclomatic complexity" | | 3. ev(g) | McCabe "essential complexity" | | 4. iv(g) | McCabe "design complexity" | | 5. n | Halstead total operators + operands | | 6. v | Halstead "volume" | | 7. l | Halstead "program length" | | 8. d | Halstead "difficulty" | | 9. i | Halstead "intelligence" | | 10. e | Halstead "effort" | | 11. b | Halstead | | | 12. t | Halstead's time estimator | | --- | --- | | 13. IOCode | Halstaed’s line count | | 14. IOComment | Halstead’s count of lines of comments | | 15. lOBlank | Halstead's count of blank lines | | 16. lOCodeAndComment | count of code and comment | | 17. uniq\_Op | unique operators | | 18. uniq\_Opnd | unique operands | | 19. total\_Op | total operators | | 20. total\_Opnd | total operands | | 21. branchCount | of the flow graph | | 22. defects | {false,true} % module has/has not one or more reported defects | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

**Correlation between features**

We have calculated the correlation among the features and represented it using a heatmap. We have observed that more than half of the features have high correlation.

For better analysis of the data we have removed highly correlated features so that we can perform analysis on the most important features.



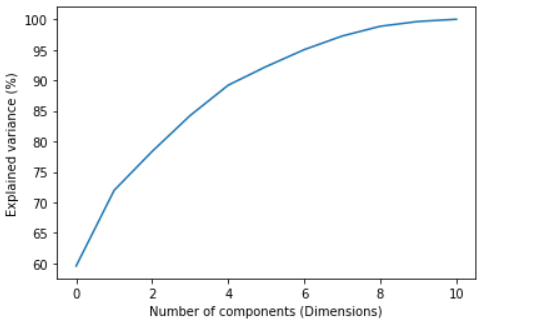
**Figure 1 : Feature Correlation heatmap**

**Removing Multicollinearity between Features**

From the above heatmap, we can say that there are many features which are highly correlated. Therefore, to remove this multicollinearity, we have applied two methods. The first method is to remove the features whose correlation is greater than 0.9. The second method we followed is to use PCA(Principal Component Analysis) which converts the set of correlated features into a set of linearly uncorrelated features called components.

Initially there are 22 features in the dataset. After applying the first method of removing multicollinearity, we are left with 12 features. We trained all the models that we specified below with this new dataset of 12 features.

For PCA, to find out the number of components that needs to be taken we plotted a graph shown below.



**Figure 2 : PCA Variance diagram**

From the above plot, we can say that by selecting 6 components, we are covering 90% of the variance in the data. Hence, we have chosen no of components = 6 and converted the dataset into a new dataset with the help of sklearn library.

**Balancing the dataset**

As mentioned the dataset has 49 true classes and 449 False classes there is a high difference between the distribution of the two classes, so this is an imbalanced dataset. So, we need to balance the dataset, so that approximately half of the distribution represents the True class and the other half represents the False class. We are balancing the dataset inorder to get appropriate results. There are a few methods to transform an imbalanced dataset into a balanced dataset in which some are mentioned below.

1. **Random OverSampler :** In this technique in order to balance the dataset random samples of data are added to the minority class so that there is not much difference between the two classes. The advantage of this technique is that there is no information loss. However there is a chance of overfitting by using this method for balancing the dataset.
2. **Random UnderSampler :** This technique involves deleting rows from the majority class so that it matches the minority class. As it involves deleting observations from the original dataset there is a chance of losing some important information. However, by using this technique the runtime for training the dataset can be improved.
3. **EditedNearestNeighbours :** The three closest neighbors for each instance an in the dataset are calculated. (a) is eliminated from the dataset if it is determined that the three closest neighbors of a majority class instance (a) misclassified a. The majority class instances among a's neighbors are also eliminated if an is a minority class instance and is incorrectly identified by its three closest neighbors.
4. **SMOTE(synthetic minority oversampling technique) :** One of the most popular oversampling techniques to address the imbalance issue is SMOTE (synthetic minority oversampling technique). By increasing minority class samples at random and duplicating them, it seeks to balance the distribution of classes. SMOTE creates new minority instances by combining minority instances that already exist.

**Methods**

We have trained 6 classification models to train the dataset. They are Logistic regression, Naive Bayes, K-Nearest Neighbor, Decision Tree, Random forest and Gradient boosting. In all these models we have GridSearchCV to train the best hyperparameters and have done cross validation with no of folds = 5.

The hyper parameters that we used to train each model are:

Logistic Regression : C = [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

Naive Bayes : var\_smoothing = np.logspace(0,-9, num=100)}

K-Nearest Neighbor : n\_neighbors' : [5,7,9,11,13,15]}

Decision Tree : max\_depth = range(1, 20) , min\_samples\_leaf = range(2, 20),

criterion = [gini, entropy]

Random Forest : n\_estimators': [200, 400, 600], max\_depth = [10, 20, 30],

Bootstrap = [True, False]

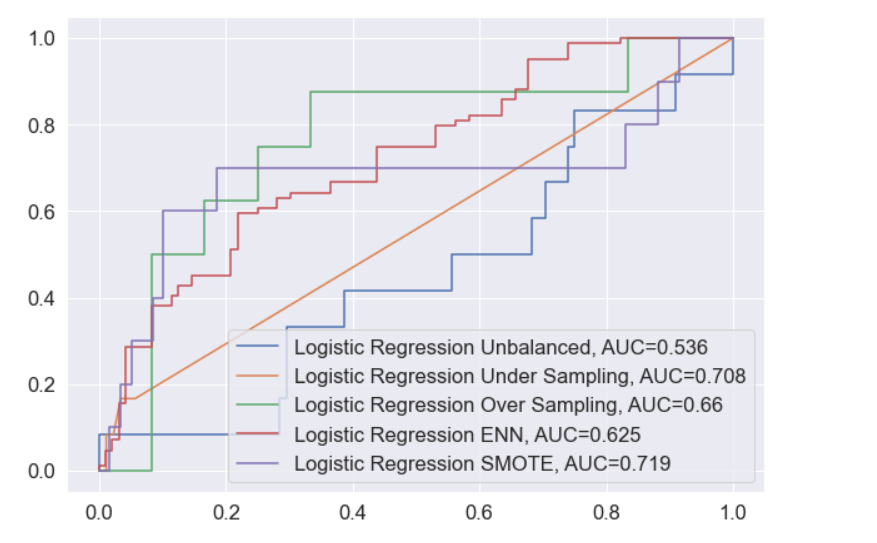
Gradient Boosting : n\_estimators = [10, 50, 100, 500], learning\_rate = [0.0001, 0.001,

0.01,0.1, 1]

**Logistic Regression :** A statistical analysis method called logistic regression uses previous observations from a data set to predict a binary outcome, such as yes or no.

By examining the correlation between one or more already present independent variables, a logistic regression model forecasts a dependent data variable. A logistic regression could be used, for instance, to forecast whether a candidate for office will win or lose, or if a high school student will be accepted into a particular institution or not. These simple choices between two options allow for binary outcomes.

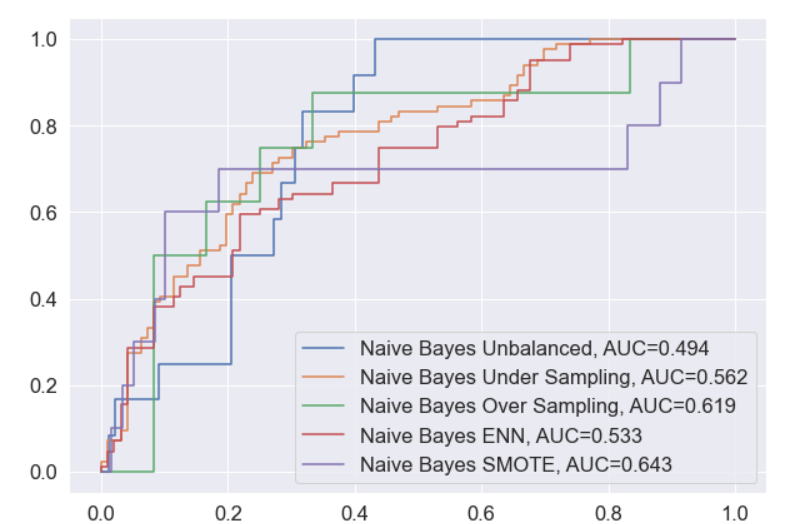
From the below plotted ROC curve, we can say that Logistic Regression with SMOTE data performs better than other models. It got AUC score 0.719 which means doing average job in distinguishing the classes.

****

**Figure 3: Logistic Regression ROC Curves**

**Naive Bayes :** Naive Bayes is a machine learning model that is used for vast amounts of data; it is the suggested strategy even when working with data that has millions of data records. It does NLP tasks like sentimental analysis with excellent results. A quick and simple classification algorithm is used. A group of classification algorithms built on the Bayes' Theorem are known as naive Bayes classifiers. It is a family of algorithms rather than a single method, and they are all based on the idea that every pair of features being classified is independent of the other.

From the below plotted ROC curve, we can say that Naive Bayes with SMOTE data performs better than other models. It got AUC score 0.643 which means doing below average job in distinguishing the classes.

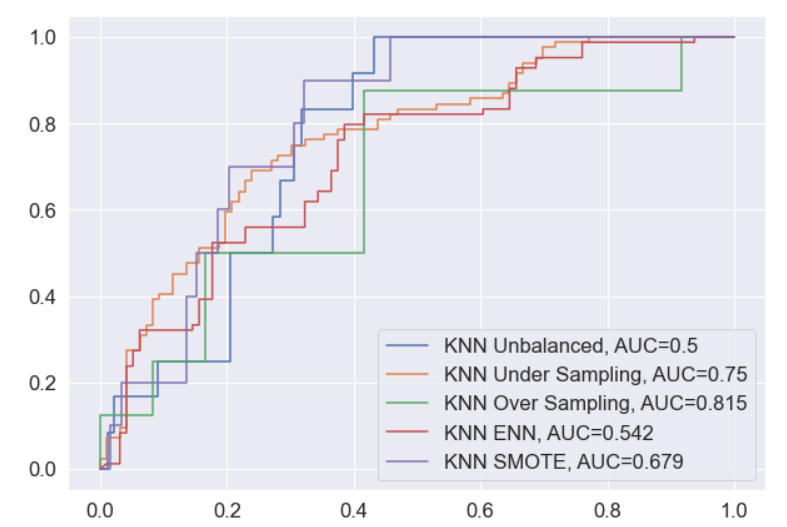


**Figure 4: Naive Bayes ROC Curves**

**K-Nearest Neighbour :** K-nearest neighbors (kNN) classifier is a supervised machine learning algorithm that is non-parametric. It is distance-based and categorizes things according to the classes of their close neighbors. Although it can be used for regression issues as well, kNN is most frequently utilized for classification tasks.

The number of labeled points (neighbors) taken into account for classification is indicated by the parameter k in kNN. The number of these points needed to calculate the outcome is indicated by the value of k. Calculating the distance and determining which categories are closest to our unidentified thing are the tasks at hand.

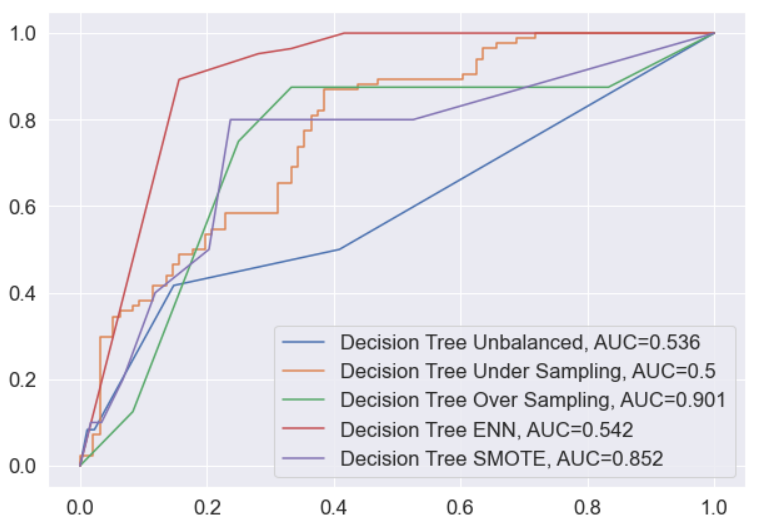
From the below plotted ROC curve, we can say that KNN with Over Sampling balancing performs better than other models. It got AUC score 0.815 which means doing below good job in distinguishing the classes.



**Figure 5: KNN ROC Curves**

**Decision Tree :** Using a branching, tree-like shape, decision trees are a popular approach to depict the decision-making process. The majority of machine learning decision trees will be applied to classification issues, classifying objects according to learned attributes. The method can be applied to regression issues or as a way to forecast continuous outcomes from unobserved data. The key advantages of employing a decision tree in machine learning are its simplicity and ease of visualizing and comprehending the decision-making process. Pruning the tree structure is frequently required because decision trees in machine learning can produce unnecessarily complicated branches by creating very granular branches.

From the below plotted ROC curve, we can say that the Decision tree with Over Sampling data balancing performs better than other models. It got AUC score 0.901 which means doing excellent job in distinguishing the classes.

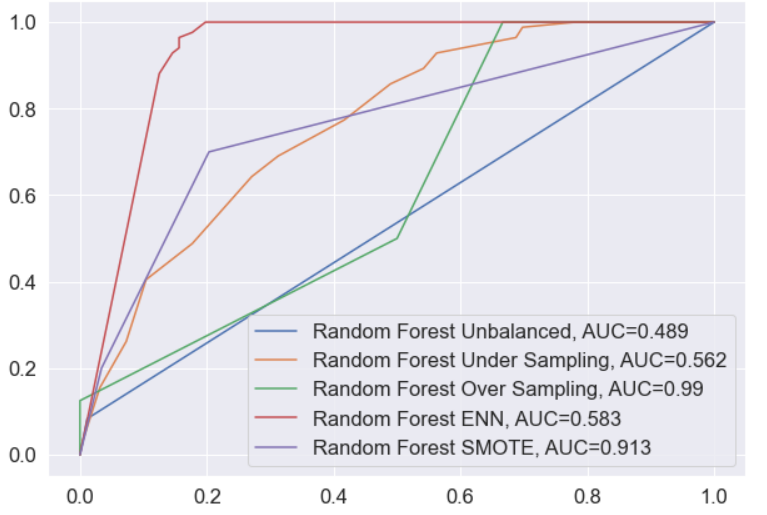


**Figure 6: Decision Tree ROC Curves**

**Random Forest :** A random forest is a machine learning method for handling classification and regression problems. It makes use of ensemble learning, a method for solving complicated issues by combining a number of classifiers.

In a random forest algorithm, there are many different decision trees. Based on the predictions of the decision trees, the random forest algorithm determines the result. It makes predictions by averaging or averaging out the results from different trees. The accuracy of the result grows as the number of trees increases.

From the below plotted ROC curve, we can say that the Random Forest classifier with Over sampling data balancing performs better than other models. It got AUC score 0.99 which means doing excellent job in distinguishing the classes.

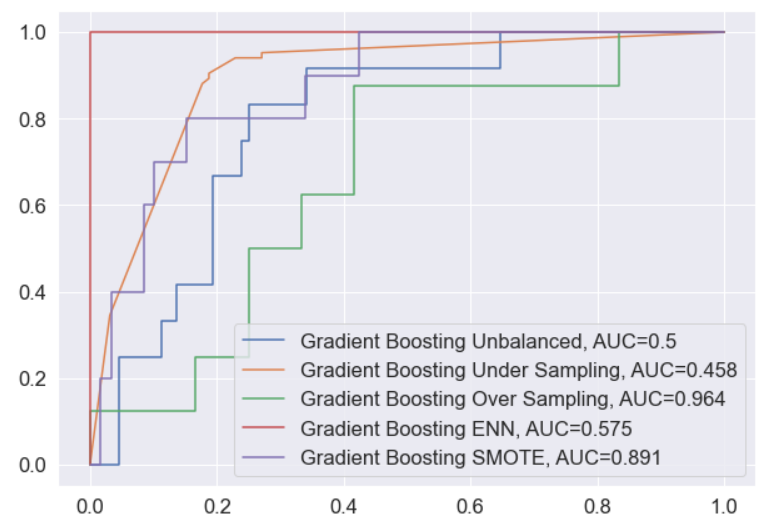


**Figure 7: Random Forest ROC Curves**

**Gradient Boosting :** One of the most effective algorithms used in machine learning is the gradient boosting technique. As is well known, bias and variance mistakes can be used to broadly categorize errors in machine learning systems. As one of the boosting strategies, gradient boosting is used to reduce the model's bias error.

Both continuous and categorical target variables can be predicted using the gradient boosting approach (as a Regressor) (as a Classifier).

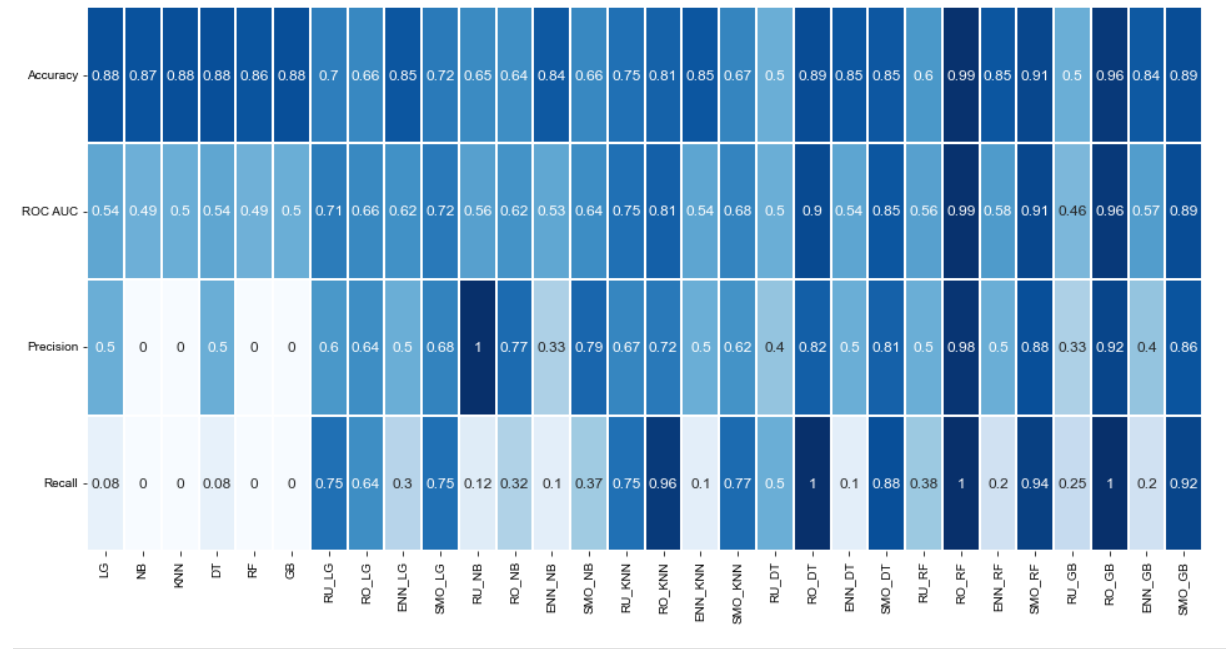
From the below plotted ROC curve, we can say that Gradient Boosting with Over Sampling data balancing performs better than other models. It got AUC score 0.964 which means doing excellent job in distinguishing the classes.

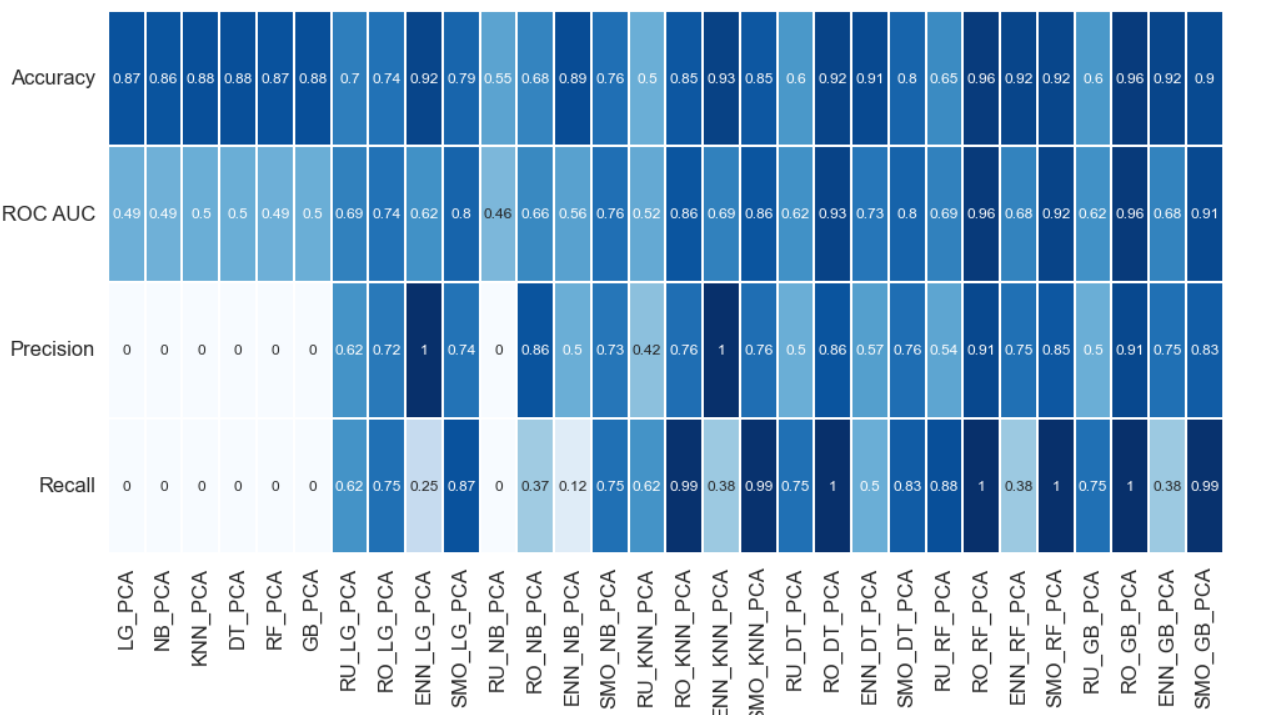
****

**Figure 8: Gradient Boosting ROC Curves**

**Visualizing Results**

We will now visualize the output obtained from each model. Plotting Accuracy, ROC AUC, Precision and recall of each model





**Conclusion**

From the above two figures, we can say that the models trained on unbalanced data do not perform well as precision and recall of those models are very low. After balancing the data with those four techniques, the precision and recall of the models increased significantly. The best model that performed with the first method of removing correlation is Random Forest with Random oversampling data. It delivered 0.99 Accuracy score, 0.99 ROC AUC Score, 0. 98 precision score and 1 recall score. Random Oversampling technique performed better than other data balancing techniques most of the time. The first method of removing multicollinearity and PCA both performed almost similarly in most of the models. For some models, PCA performed better and for other models the 1st method of removing collinearity performed better.

**References:**

1. Fenton, N.E. and Neil, M., 1999. A critique of software defect prediction models. *IEEE Transactions on software engineering*, *25*(5), pp.675-689.
2. Li, Z., Jing, X.Y. and Zhu, X., 2018. Progress on approaches to software defect prediction. *Iet Software*, *12*(3), pp.161-175.
3. Wahono, R.S., 2015. A systematic literature review of software defect prediction. *Journal of Software Engineering*, *1*(1), pp.1-16.
4. Okutan, A. and Yıldız, O.T., 2014. Software defect prediction using Bayesian networks. *Empirical Software Engineering*, *19*(1), pp.154-181.
5. Thota, M.K., Shajin, F.H. and Rajesh, P., 2020. Survey on software defect prediction techniques. *International Journal of Applied Science and Engineering*, *17*(4), pp.331-344.
6. Li, J., He, P., Zhu, J. and Lyu, M.R., 2017, July. Software defect prediction via convolutional neural network. In *2017 IEEE international conference on software quality, reliability and security (QRS)* (pp. 318-328). IEEE.
7. Lessmann, S., Baesens, B., Mues, C. and Pietsch, S., 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, *34*(4), pp.485-496.
8. Wang, S. and Yao, X., 2013. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, *62*(2), pp.434-443.
9. Challagulla, V.U.B., Bastani, F.B., Yen, I.L. and Paul, R.A., 2008. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools*, *17*(02), pp.389-400.
10. Shepperd, M., Bowes, D. and Hall, T., 2014. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, *40*(6), pp.603-616.
11. Nam, J., 2014. Survey on software defect prediction. *Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Tech. Rep*.
12. Wang, Q., Wu, S. and Li, M.S., 2008. Software defect prediction. *Journal of software*, *19*(7), pp.1565-1580.
13. Laradji, I.H., Alshayeb, M. and Ghouti, L., 2015. Software defect prediction using ensemble learning on selected features. *Information and Software Technology*, *58*, pp.388-402.
14. Ma, Y., Luo, G., Zeng, X. and Chen, A., 2012. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, *54*(3), pp.248-256.
15. He, P., Li, B., Liu, X., Chen, J. and Ma, Y., 2015. An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology*, *59*, pp.170-190.
16. Pelayo, L. and Dick, S., 2007, June. Applying novel resampling strategies to software defect prediction. In *NAFIPS 2007-2007 Annual meeting of the North American fuzzy information processing society* (pp. 69-72). IEEE.
17. Li, N., Shepperd, M. and Guo, Y., 2020. A systematic review of unsupervised learning techniques for software defect prediction. *Information and Software Technology*, *122*, p.106287.
18. Felix, E.A. and Lee, S.P., 2017. Integrated approach to software defect prediction. *IEEE Access*, *5*, pp.21524-21547.
19. Chen, X., Gu, Q., Liu, W.S., Liu, S.L. and Ni, C., 2016. Survey of static software defect prediction. *Journal of Software*, *27*(1), pp.1-25.
20. Wang, T. and Li, W.H., 2010, December. Naive bayes software defect prediction model. In *2010 International conference on computational intelligence and software engineering* (pp. 1-4). Ieee.
21. Rodriguez, D., Herraiz, I., Harrison, R., Dolado, J. and Riquelme, J.C., 2014, May. Preliminary comparison of techniques for dealing with imbalance in software defect prediction. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (pp. 1-10).
22. Qiao, L., Li, X., Umer, Q. and Guo, P., 2020. Deep learning based software defect prediction. *Neurocomputing*, *385*, pp.100-110.
23. Bowes, D., Hall, T. and Petrić, J., 2018. Software defect prediction: do different classifiers find the same defects?. *Software Quality Journal*, *26*(2), pp.525-552.
24. Rawat, M.S. and Dubey, S.K., 2012. Software defect prediction models for quality improvement: a literature study. *International Journal of Computer Science Issues (IJCSI)*, *9*(5), p.288.
25. Jing, X.Y., Ying, S., Zhang, Z.W., Wu, S.S. and Liu, J., 2014, May. Dictionary learning based software defect prediction. In *Proceedings of the 36th international conference on software engineering* (pp. 414-423).
26. Matloob, F., Ghazal, T.M., Taleb, N., Aftab, S., Ahmad, M., Khan, M.A., Abbas, S. and Soomro, T.R., 2021. Software defect prediction using ensemble learning: A systematic literature review. *IEEE Access*.
27. Li, M., Zhang, H., Wu, R. and Zhou, Z.H., 2012. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, *19*(2), pp.201-230.
28. Dam, H.K., Pham, T., Ng, S.W., Tran, T., Grundy, J., Ghose, A., Kim, T. and Kim, C.J., 2018. A deep tree-based model for software defect prediction. *arXiv preprint arXiv:1802.00921*.
29. Liang, H., Yu, Y., Jiang, L. and Xie, Z., 2019. Seml: A semantic LSTM model for software defect prediction. *IEEE Access*, *7*, pp.83812-83824.